

INFORMATION RETRIEVAL IN INDIAN LANGUAGES

by

K. RAJA SEKHARA REDDY

TN
CSE/1995/m
R2461

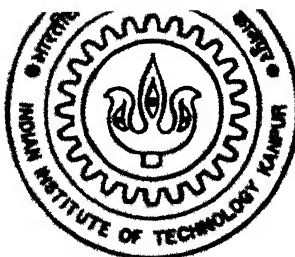
CSE

1995

M

RED

INF



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
JANUARY, 1995

Information Retrieval in Indian Languages

*A dissertation submitted
in conformity with the requirements
for the degree of*

Master of Technology

by

K. Raja Sekhara Reddy

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

January 1995

13 FEB 1995/CSE

CN MARY

Case No A .118778

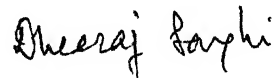


A118778

CSE-1995-M-RED-INF

Certificate

It is certified that the work contained in this thesis, entitled Information Retrieval in Indian Languages, has been carried out by K. Raja Sekhara Reddy (Roll No: 9311108) under my supervision, and that this work has not been submitted elsewhere for a degree.



Dr. Dheeraj Sanghi

Assistant Professor,
Department of Computer Science and
Engineering
IIT, Kanpur

January 1995

Synopsis

Name of Student : *K. Raja Sekhara Reddy*
Roll No : *9311108*

Degree for Which Submitted : *M.Tech.*
Department : *Computer Science and Engineering*

Thesis Title : *Information Retrieval in Indian Languages*
Name of Thesis Supervisor : *Dr. Dheeraj Sanghi*

Month and Year of Submission : *December 1994*

Thesis Abstract :

In a full-text natural language information retrieval system, the need for support language processing tools arises, e.g., for content analysis in generating index terms for documents. Such tools are highly desirable for Indian languages which have complex morphological and syntactic structure. By developing these language processing support tools, we can use any existing document retrieval system for Indian languages also. Some simple and practical tools for necessary linguistic support are designed and implemented. These tools are then used to extend a well-known distributed information retrieval system, Wide Area Information Servers(WAIS) to support natural language text retrieval in Telugu. The developed tools are general and can be used for any Indian language.

Key-words and phrases : *Information retrieval, language processing, content analysis, stemming, WAIS, Indian languages, Telugu.*

Acknowledgements

I am extremely grateful to my supervisor, *Dr. Dheeraj Sanghi*, for the time, patience and insight he has given over the past 12 months and for being available to discuss problems and answer any questions. I would also like to thank him for the prompt reading of the drafts of this thesis, and for many suggestions and criticisms that led to a better presentation of the material.

I am indebted to *Dr. Rajeev Sangal* for the profitable discussions I have had with him, and for his help in various forms at all stages of this work. I would also like to thank my external examiner *Dr. S. Sadagopan* for his valuable suggestions and comments.

Thanks are due to all my friends and classmates who made my brief stay at IIT Kanpur a memorable one. The most important one to name, on this occasion, is *Uzzal*. I am very thankful to him for his help in various academic tasks, including this thesis work, as well as his moral support throughout these days.

I would also like to utilize this opportunity to thank my father-in-law *Mr. P. Panchala Reddy* who motivated me to join M.Tech. Finally, I thank my parents and my wife for their continued cooperation during these days apart.

Jan 10, 1995

K. Raja Sekhara Reddy

Age does not protect you from love.

But love, to some extent, protects you from age.

- *Jeanne Moreau*

To my wife
Kranthi

Contents

Chapter 1	Introduction	1
1.1	Document Retrieval	2
1.1.1	Components of a Document Retrieval System	2
1.1.2	Performance Indices	4
1.2	Language Processing in Document Retrieval	5
1.3	Outline of This Work	6
Chapter 2	Content Analysis	10
2.1	Overview of the Process	10
2.2	Common Word Deletion	13
2.3	Synonym Lookup	13
2.4	Phrase Recognition	14
Chapter 3	Stemming Algorithm	17
3.1	Words and Word Formation	17
3.2	Stemming Algorithm	19
3.3	Suffix Truncation	20
3.4	Stem Recoding	22
3.5	Lexicon	25
3.6	Spelling Checking	27
Chapter 4	Implementation	29
4.1	Stemming	29
4.1.1	Suffix Dictionary	30
4.1.2	Rule-based Recoding	31

4.1.3	Recoding by Stem Ending Replacement	37
4.1.4	Exceptional Words List	38
4.2	Lexicon	39
4.3	Others	41
 <i>Chapter 5 WAIS to Support Document Retrieval in Telugu</i>		43
5.1	An Overview of WAIS	43
5.2	Integration	46
 <i>Chapter 6 Conclusions</i>		48
6.1	Summary of This Work	48
6.2	Scope for Further Work	49
 Bibliography		52
 <i>Appendix A The Encoding Scheme</i>		55
 <i>Appendix B Sample Dictionaries</i>		56
 <i>Appendix C Important Modules</i>		66
C.1	Suffix Trie Routines	66
C.2	NFA Construction	68
C.3	NFA to DFA Conversion	68
C.4	Running the DFAs	72
C.5	Recoding by Stem Ending Replacement	73
C.6	The Hash Function	74

List of Figures

1	A Typical Document Retrieval System	2
2	Dynamics of a Document Retrieval System	3
3	The Content Analysis Procedure	11
4	An Example of Word Formation in Telugu	18
5	Steps Involved in the Stemming Algorithm	20
6	The Stemming Algorithm	21
7	Examples Showing the Need for Stem Recoding	23
8	Possibility of Recoding by Listing out Equivalent Stems	23
9	Spelling Checker	27
10	The Compiler and Analyzer components of the stemmer	30
11	An example of inverse suffix trie	31
12	Grammar of Recoding Rules Specification	32
13	Lexicon - (a) Compilation: Building the Check Hash List (b) Analysis: Searching the Lexicon	40
14	Phrase Detection and Matching	42
15	The Encoding Scheme for Telugu	55
16	Constructing an NFA from Input Rules Specification	68

Introduction

Information Science is an emerging field that deals with storage, retrieval and transmission of all types of information. The most important computer-based information systems today are the management information systems (MIS), database management systems (DBMS), decision support systems (DSS), question-answering systems (QA) and information retrieval systems (IR). In this thesis, we are mostly interested in information retrieval or document retrieval, which deals with the representation, storage and access to documents.

The problem of document retrieval is this: there exist a store of documents and the user formulates a query to which the answer is a set of documents satisfying the information need expressed in that query. To judge which of the documents in the document store satisfy the user's need, the system performs some analysis of these documents. This process of analysis, called *content analysis* or *subject indexing*, involves formulating a set of keywords that describe the content of the given document or query. These keywords are then used for determining which of the documents are relevant to the user's request.

Content analysis involves little language processing because of the fact that both the documents and the user requests consist of natural language text. By designing tools for the required language processing, the existing document retrieval systems, most of which assume English as the language of the text, can be used for Indian languages also. There is a large amount of research going on in computer processing of Indian languages [6]. The result of these works may not be of direct use to us because of the fundamental differences in the (linguistic) requirements of document retrieval and of the other areas of language processing.

The major aim of this work is to provide the conceptual framework and working tools needed to satisfy various linguistic requirements in document retrieval systems for Indian languages. These tools are then used to extend a well-known distributed information retrieval system, WAIS (Wide Area Information Servers), to support document retrieval in Telugu. Although the problems discussed, the suggested solutions, and the tools implemented are specially tailored for Indian Languages, the general ideas can be applied to other languages with similar characteristics.

Rest of this chapter presents necessary background material on document retrieval and

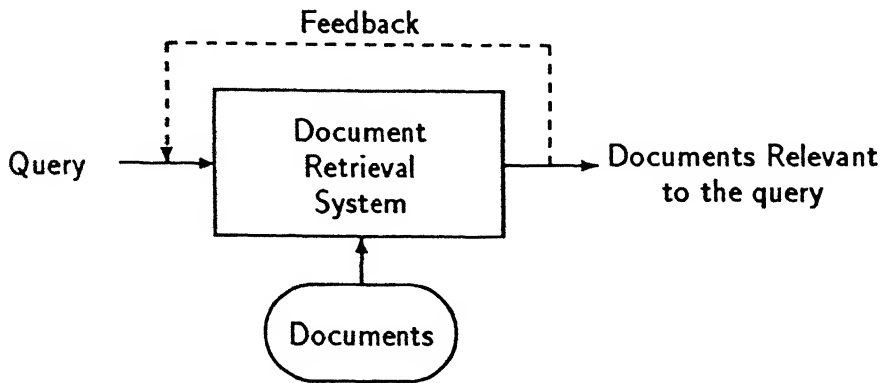


Figure 1: A Typical Document Retrieval System

examines the role of language processing in document retrieval. The formal definitions and issues involved in the problem we have undertaken are provided at the end of the chapter.

1.1 Document Retrieval

Document Retrieval is the problem of finding stored documents that contain useful information. Figure 1 illustrates by means of a black box how a typical document retrieval system functions. The system consists of a number of documents. User queries the system to search for recorded information that may be contained in some of the documents in the set. In each instance in which an individual seeks information, he or she will find some documents of the set useful and other documents not useful; the documents found useful are, we say, *relevant*; the others, *irrelevant*. The problem is to retrieve *all* the documents relevant to a given query and at the same time, retrieving *only* the relevant.

To determine which documents should be retrieved in response to a query, a set of *keywords* or *index terms* are first produced from the documents and queries. The set of all index terms used to represent documents or queries in a given document store constitutes the *indexing vocabulary*. The indexing vocabulary is either prespecified (*controlled*) or taken freely from the text of the documents and queries (*uncontrolled*). Each information item (a document or a query) is thus represented by a list of elements from the indexing vocabulary, called a *descriptor* or *representative*. The procedures for identifying which documents are relevant to a given query are based on the representatives of the documents and queries.

1.1.1 Components of a Document Retrieval System

Figure 2 shows various components of a typical document retrieval system and their interaction. A brief description of each of these components follows:

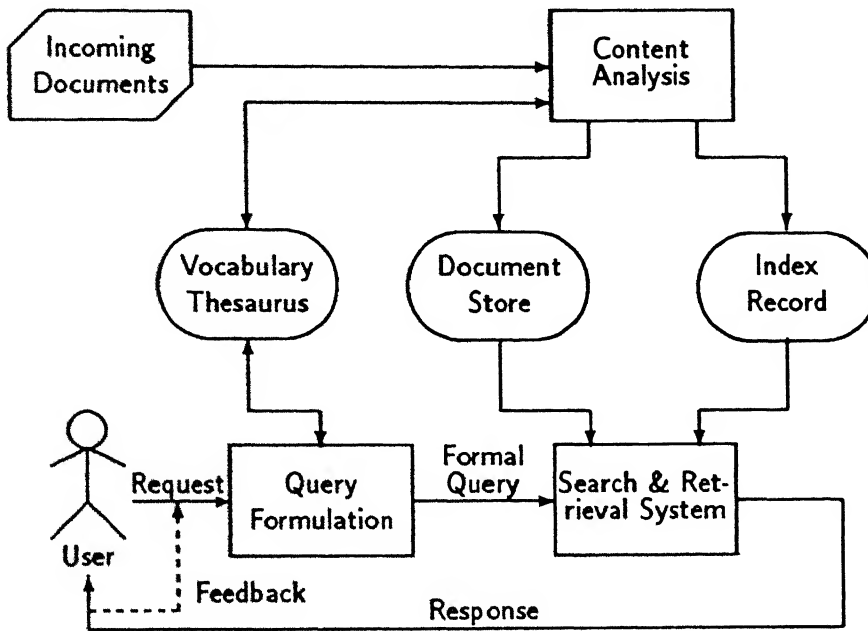


Figure 2: Dynamics of a Document Retrieval System

User and Incoming Documents These two blocks serve as points of input to the system; they define the environment in which the retrieval system operates.

Document Store The document store is the place where the documents are stored for later retrieval. Some retrieval systems may store only *surrogates* of the documents in the document store, which means that the text of a document is lost once it has been processed for the purpose of generating its representatives. A document surrogate could be a title, an abstract, an extract or even the complete document itself.

Content Analysis The incoming documents first undergo a process of content analysis and representation. The primary function of this block is to extract two outputs from the incoming documents:

- a document surrogate to place in the document store.
- the content descriptor of the document in the given indexing vocabulary as determined from the vocabulary thesaurus to store in the index record along with a pointer to the document surrogate in the document store.

The generation of content descriptor for a given document is the most difficult task in the complete system. This process involves ascertaining the meaning inherent in the natural language text of the documents and expressing this in the underlying indexing vocabulary.

Index Record Following the choice of representatives for the documents in the indexing vocabulary, these representatives are organized into a file structure called index record. Some factors of interest in choosing a file structure involve the efficiency and/or effectiveness of operations such as searches and updates. The most frequently used file structure is the inverted index[34, 47].

Query Formulation User submits the queries to the system in natural language. The task of query formulation module is to recognize and transform the query into a formal query (that is a query expressed in terms from the indexing vocabulary using some logical relations). This translation process, thus, involves a limited form of content analysis also.

Search and Retrieval Component This component performs two major functions:

- It interprets the formal query and searches the index record to identify the relevant documents. It may then interact with the users to finalize the set of documents to be retrieved.
- It consults the document store to extract the required documents and presents them to the user.

User Feedback When the retrieval system is on-line, it is possible for the users to change their requests during one search session in the light of a sample retrieval, thereby improving the subsequent retrieval run. Such a process is commonly referred to as *feedback*.

1.1.2 Performance Indices

The following six main measurable quantities that will reflect the ability of the system to satisfy the user have been suggested [11, 30] –

1. the *coverage* of the document collection, that is, the extent to which the system includes relevant matter.
2. the *response time*, that is, the average interval between the time the query is formulated and the time an answer is given.
3. the form of *presentation* of the output.
4. the *effort* involved on the part of the user to obtain relevant information from the system.
5. the *recall* of the system, that is, the ability of the system to to present all relevant documents.
6. the *precision* of the system, that is, the ability of the system to present only the relevant documents.

We are mainly interested in the last two quantities, namely recall and precision, which measure the effectiveness of the retrieval system.

Let us say, D is the set of documents in a given database. Let us assign to each request q , a subset D_q of D which is the set of reference documents “relevant” to q . After the retrieval operation, let D_a be the actual set of documents retrieved. Using these sets we can define

$$Recall = \frac{|D_q \cap D_a|}{|D_q|}$$

$$Precision = \frac{|D_q \cap D_a|}{|D_a|}$$

where $|A|$ denotes the number of elements in set A .

Clearly Recall is a measure of the inclusiveness of the set D_a with respect to the set D_q , while precision is a measure of the exclusiveness of the set D_a with respect to the complement of D_q . In other words, Recall measures how well the system retrieves *all* the relevant documents and Precision, how well the system retrieves *only* the relevant documents.

It should be noted that the joint behaviour of these parameters is required to judge the performance intuitively; that is, a recall of 1 or a precision of 1 is alone does not imply satisfactory performance; however, if both recall and precision are 1, then $D_a = D_q$.

1.2 Language Processing in Document Retrieval

In this section, we examine the role of language processing methods in the document retrieval context. Three levels of language processing are of interest:

- the *morphological* level, which is concerned with words and word formation process
- the *syntactic* level, which deals with the analysis of how words are combined into structural units (such as noun phrases, propositional phrases etc.) and sentences
- the *semantic* level, which deals with actual meaning of words and sentences.

The advantages of using language processing approaches in a document retrieval system are as follows [13, 22, 46]:

Naturalness Documents are in natural languages and the person seeking information from a document collection will almost always find it easiest to put his request in the language of everyday discourse. The use of natural language search statements could raise the efficiency as well as the effectiveness of the retrieval operations by making possible the formulations of requests that correctly reflect the user needs and by simplifying the user-system interactions.

Precision A content descriptor of a document or a request which takes the form of an essentially unordered list of terms (whether or not it is chosen from a controlled vocabulary) is fundamentally imprecise. A user query dealing with the subject of computational complexity and described by the index terms `compute` and `complex` is just as likely to also cover extraneous topics such as `computation with complex numbers` besides the actual subject area of interest. If the system is sensitive to syntax, then there is a better chance of getting only what the user wants. This improves the precision of the retrieval.

Ambiguity and Synonymy. Natural language words as well as phrases and sentences are inherently ambiguous. A single word can have multiple meanings (e.g., the word `bank` may mean the ground near a river, an establishment for custody of money, or slope of a road) and different words may refer to the same thing (e.g., the words `machine` and `computer` are equivalent in the context of computer science). The problems of synonymy and ambiguity must be dealt with if an accurate content analysis of documents and queries is to be undertaken. Linguists have shown that the formal processes of syntactic and semantic analyses can help much in dealing with these problems.

1.3 Outline of This Work

Of all the operations required in information retrieval, the most crucial one consists of assigning appropriate terms and identifiers capable of representing document content. In principle, a document analysis process is redundant if the document collection is small enough to permit the scanning of the full-text of all documents, whenever a request for information is received. In practice, such a solution is too expensive in terms of processing time. Hence each document is characterized by assigning a set of keywords to the document which can be used to obtain access whenever the document is wanted.

Ideally, an automatic content analysis should duplicate the human process of ‘reading’ and ‘understanding’ to identify what the document is about and obtain a set of keywords and phrases that can describe the document content. Such a process involves the use of a complete natural language understanding system which doesn’t seem to be practical with the current state of development in language processing. However, it may be noted that the requirements in document retrieval are not that stringent; The exact meaning of document text is not important. Two documents dealing with the same subject matter but coming to different conclusions are identical, as far as the retrieval is concerned. Hence, satisfactory system performance can be obtained by approaching the problem with simplified and more practical solutions.

In this thesis, we aim at designing and implementing such a simple and practical content analysis algorithm for Indian languages. The implemented tool is then integrated into WAIS, a well-known distributed information retrieval system, to support document retrieval in Indian languages. The main issues addressed are summarized below.

Roman Encoding of Indian Language Script. To use the current roman keyboards to input the Indian language text, we first need to design a scheme for encoding Indian

language letters using the Roman letters. We adapt here the encoding scheme used in Anusaraka [6]. This scheme has some important features that enables it faster to learn and easier to use:

- It is general and can be used for any Indian language.
- It uses only the small case and capital case Roman letters for representing Indian language letters. It doesn't use digits or special symbols.
- Most of the Indian language letters are represented by a single Roman letter. The few cases that use two Roman letters to encode a single Indian language letter are easy to remember because they are based on certain simple rules.
- The representation scheme is based on certain phonetic principles in most of the cases. There are very few exceptions which are to be memorized.

The representation scheme for Telugu is shown in Appendix A.

Morphological Analysis. Morphology, or the study of word formation, provides us with the safest entry into the exploration of natural language. Words are formed out of roots and affixes. For example, the English word *discontinued* is formed from the root *continue* by attaching the prefix *dis-* and the suffix *-ed*. Morphological analysis is the process by which, given a word, its root and other grammatical information are obtained.

Morphological analysis is an extremely useful step in generating content identifiers for documents and queries [2, 13, 22, 31, 32]. This reduces a variety of different forms such as *analysis*, *analyzing*, *analyzer*, *analyzed* and *analysing* to a common root 'analyze'. The word stem *analyze* will have a higher frequency of occurrence in the document texts than any of the variant forms. Hence, when the word roots or stems are used as indexing terms, a greater number of potentially relevant documents can be identified, for a given query, than one of the full text words is in use. This serves, thus, as a recall enhancing device.

Languages differ radically in the complexity of morphological processes they employ. Morphology is relatively simple in English. Indian languages have a rich and complex morphology. One important contribution of this work is a general algorithm for morphological analysis of Indian languages.

Syntactic Analysis. Syntactic analysis is useful in assigning complex index terms like phrases replacing the single terms that are normally used, thereby enhancing the index process. The term *programming language*, for instance, instead of the separate terms *programming* and *language* avoids mismatches with texts on programming for natural language processing through the implicit syntactic structure of the term. Many limited syntactic analysis procedures which aim at identifying such sentence elements considered most relevant for content analysis have been suggested [13, 22]. These procedures essentially identify syntactic word groups like noun phrases, verb phrases, propositional phrases

etc. No attempt is made to use such syntactic analysis procedures in this work. Main reasons for it may be summarized as follows:

- Most of the suggested schemes were based on dictionaries supplying syntactic information on words. These dictionaries are relatively difficult to supply and processing may also be slower.
- Many earlier major experiments like those carried out by the SMART Project[42], Comparative Systems Laboratory[12], and Lancaster[29] have shown that syntactic tools are of little value. On the other hand, some simple semi-automatic procedures that do not use any syntactic information for phrase recognition, like the one suggested later in this dissertation, give better results.
- The usefulness of such syntax-based phrase recognition schemes for Indian Languages is not yet well understood. For example, the concept of verb phrases doesn't seem to be natural for Indian Languages [6].

In this dissertation, we use a phrase recognition scheme based on a priori dictionary of phrases. This scheme doesn't make use of any syntactic information from the text of the document and doesn't depend on syntactic dictionaries.

Semantic Analysis. In content analysis of documents, semantic analysis is useful in resolving ambiguities and recognizing synonymies that exist in the natural language text. This helps in generating conceptual categories as index terms than text words or word stems. There has been a great deal of interest in semantic processing, although efforts have been largely limited to various types of statistical analysis than pure linguistic based approaches [13, 22, 43]. Such statistical techniques are generally based on co-occurrence characteristics of terms in the documents of a particular document collection.

To give a simple example, consider a document collection $D = \{D_1, D_2, \dots, D_n\}$ of n documents. Consider two terms a and b with term frequency vectors $(f_{1a}, f_{2a}, \dots, f_{na})$ and $(f_{1b}, f_{2b}, \dots, f_{nb})$, where f_{ix} denotes the frequency of the term x in the document D_i . A simple similarity measure may be defined as[46]

$$SIMILAR(a, b) = \sum_{i=1}^n f_{ia} f_{ib}$$

Two terms a and b may, then, be considered as 'synonyms', as far as the retrieval operation is concerned, if $SIMILAR(a, b)$ is sufficiently large.

The problem with such statistical approaches is that the applicability of the resulting synonym thesaurus is restricted to a given document collection than to a complete subject field. As a result, when few more documents have to be added to the collection, the synonym thesaurus may change, and all the documents may have to be re-indexed. And also, such procedures, due to lack of any linguistic support, may falsely conclude that the words *computer* and *network* are synonyms, and may fail even to regard singular and plural forms as the occurrences of the same word.

For this reason, it is of interest to consider other approaches which involve some amount of human judgement. In such methods, synonym recognition or ambiguity resolution is based on a synonym dictionary supplied by subject experts. One such approach has been used in this work.

Organization of this thesis

Chapter 2 describes the implemented content analysis procedure. One major step in the process of content analysis is morphological analysis or stemming. We look into the issues and details of the proposed stemming algorithm for Indian languages in Chapter 3. Chapter 4 gives an account of various issues involved in implementing the suggested content analysis algorithm. The important data structures used in implementation and the procedures for some major components of the algorithm are summarized. Chapter 5 discusses how the implemented content analysis tool for Indian languages is used to extend WAIS to support document retrieval in Telugu. The main problems faced in integration and their solutions are listed. Finally, Chapter 6 summarizes the contributions of this work, and suggests possibilities for future work in this area.

Content Analysis

In this chapter, we describe the content analysis process for Indian languages. Content analysis is the process by which, to each document or query, a list of terms which describe the most important concepts of that document or query, are assigned. During the index construction process, these terms are used to build an inverted file of indices; during the retrieval, the extracted terms from the query are used to identify the relevant documents by matching the keywords to those in the index. This operation is very important because the retrieval effectiveness of the system completely depends on the index terms assigned to the documents.

Below, a brief description of the content analysis process is first provided and then each of the steps involved are described in more detail.

2.1 Overview of the Process

The simplest possible word indexing scheme is to use the set of all words in a document as its content descriptor. This form of indexing is easy to apply and doesn't require any subject matter background or language knowledge. But the retrieval performance would be poor. To improve the performance of the word indexing scheme, some features for filtering and normalizing the text words are added. The resulting scheme is a five step process as described below (see Figure 3):

Step 1 – *Extract sentences*

The document text is processed sentence by sentence. Hence the first step is to isolate individual sentences from the document text. This is done through defining a sentence as a sequence of words that occur between two full-stops(.). This simple-minded strategy creates problems when fullstops are used for some other purposes like in abbreviations. But in this work, we will ignore these problems. Each sentence is then subjected to the following sequence of steps.

Step 2 – *Delete common words*

Some words in the text are meant to carry some syntactic functions. Such words are

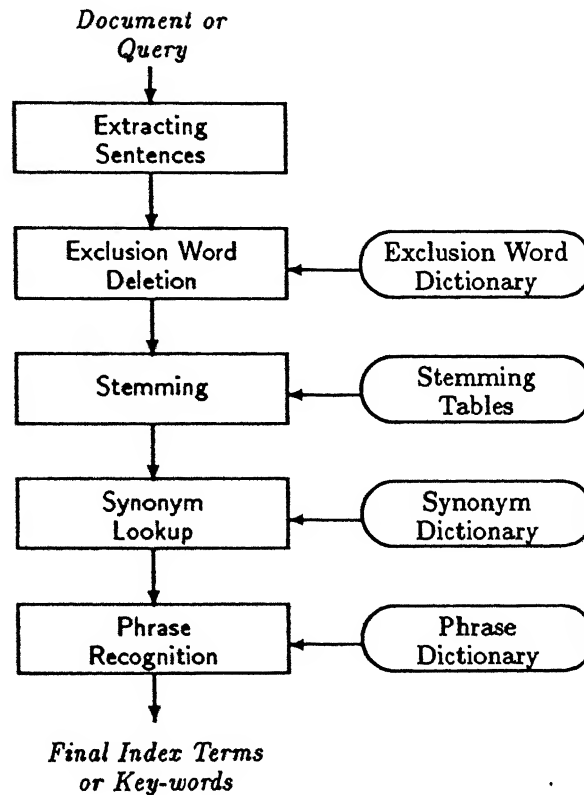


Figure 3: The Content Analysis Procedure

insignificant because they do not directly contribute to the specification of information content. For example, English words like *for*, *is*, *are*, *the*, *as* etc. Such words should not be included in the machine-generated index for the following reasons:

- This reduces the number of words to be indexed greatly.
- This improves the retrieval precision for the following reason. Such words presumably are the most frequently occurring in a given text. So, a query that contains these words results in the retrieval of lot of unrelated documents, thereby leading to lower precision value.

Thus, the first step, after sentences are isolated, is to delete such common function words from the given sentence. This is done using a dictionary of such words, sometimes called a *negative dictionary* or *stop list*.

Step 3 – Transform words to stems

The next step, following the removal of stop words, is to transform all words in the sentence to their basic stem forms. In many cases, the information which is semantically significant to the user is contained in the stems of the lexical words in the document terms, and suffixes merely enable this information to be expressed in a grammatical form. The form of the words which the user inputs may not correspond to that of the original words in the index. To permit the words in the user's query to match the words in the index entry's terms, both query and document terms can

be stripped of the suffixes that prevent their matching. For example, *computing*, *computer*, *computers*, *computational*, and *computerization* might all be stemmed to *comput*.

This process, called morphological analysis or stemming¹, provides one way of grouping related words together to increase the number of matching query and document terms, thereby enhancing the retrieval recall.

Step 4 – *Replace stems by concepts*

Many words may be used to supply the same or related meanings. The same words that are used in a document may not appear in the query of a user who wants to access that document. Such synonymous words must be recognized if an accurate content analysis of documents and search requests must be undertaken. Also, by recognizing synonyms, we can generate additional matches between query and document terms, thereby obtaining more documents and improving the recall output.

This synonym detection process uses a synonym dictionary in which stems are grouped into synonym or affinity classes. Each class is given a concept identifier. In this step, stems are replaced by the identifiers of the concept classes to which they belong.

Step 5 – *Detect Phrases*

Consider the phrase *programming languages*. A query that has this phrase may result in retrieving the documents that discuss about programs or programming, the documents that discuss about natural languages or documents that deal with programs for natural language processing along with the documents that contain something on programming languages. This reduces the precision of the retrieval. This is because of the relatively more ambiguity in the terms *program* and *language* standing alone than the phrase *programming language*.

Moreover, many types of syntactic equivalences occur in the language, where completely different constructions are used to represent the same general idea; for example the phrases *programming languages* and *languages for programming* both mean the same. And it is false to assume that the users can guess the exact phrases being used in the documents. So it is important to recognize at least the principal types of paraphrasing.

In this last step, possible phrases in the sentence are identified. The phrase recognition process uses a dictionary of phrases which is prepared by the subject expert. Identifiers of the detected phrases are appended to the sentence.

At the end, the sentence in its final shape, contains all the index terms generated from the original sentence. Index terms of all sentences together constitute the content descriptor for the document. Each of the four steps – common word deletion, stemming, synonym lookup and phrase detection – help improve the retrieval effectiveness. While the synonym and stemming facilities improve recall, common word deletion and phrase detection improve precision.

¹The word *stemming* is used as synonym to *morphological analysis* throughout this dissertation.

The three steps, common word deletion, synonym lookup and phrase detection, are discussed in detail in the following sections, while the more complex stemming process is described in the next chapter.

2.2 Common Word Deletion

Almost all existing retrieval systems make some provisions for removal of certain words that are believed not useful for content identification. Such list of stop words normally includes function words like prepositions, articles, conjunctions, auxiliary verbs, etc. These comprise upto 30 to 40 percent of the text. Elimination of these words improves the retrieval precision and saves space by reducing the indexing vocabulary. Apart from such function words, certain very common words for a particular database may also be included in the list. For example, in a document collection dealing with computer science, such words as *machine* and *computer* occur in most documents and hence cannot be good discriminators.

This facility is simple to provide through the use of an *Exclusion Word Dictionary* (EWD). Frequently, a standard EWD is provided with the document retrieval package. The system manager can add terms specific to the database. Some principal guidelines to be followed when constructing the EWD may be enumerated as follows:

1. Very common high frequency terms should be included in the Exclusion Word Dictionary, since they produce too many matches for effective retrieval.
2. Nonsignificant high frequency words should be carefully studied before they are included in the EWD. For example, a term such as *hand* should not be included in the EWD dealing with biology, but it should be included if its high frequency count is due to expressions such as "on the other hand".
3. One may include words that occur very rarely in the database (say, once in the whole document collection), since they could not be expected to produce many matches between documents and queries. But this should be done with care because those few documents dealing with these words may not be retrievable.

2.3 Synonym Lookup

The synonym lookup step involves recognizing equivalent stems occurring in the text and choosing those stems to be used as index terms. This is performed using a dictionary called *Synonym Dictionary* (SD).

Synonym dictionary provides a grouping of word stems into certain subject categories or concept classes. Many different word stems may map into the same concept class. There are some words that can have several meanings, depending on the context (for example, the word *base* may variously represent military bases, lamp bases, bases in the baseball etc.). For such ambiguous words, entries are made in many concept classes to allow all

possible meanings. Thus the SD performs a many-to-many mapping from word stems to concept classes.

Synonym dictionary suitable for the given context is provided by the database manager. While constructing a synonym dictionary, one principal problem that arises is what type of synonym categories should be created – that is, should one aim for broad, inclusive concept classes, or should the classes be narrow and specific [43]. It is clear that if very broad and somewhat fuzzy categories are wanted, the resulting dictionary will interpret a question in a reasonably broader sense, and as a result the recall will be rather high. At the same time, the precision may be low, since much irrelevant may also be produced in the process. If, on the other hand, the categories are specific, the chance of picking up irrelevances is much smaller and, therefore, the precision is increased; however, the recall may suffer, since relevant matter is likely to be missed at the same time.

So the synonym dictionary should be constructed on the basis of required characteristics of the retrieval environment. If the retrieval needs to be reasonably complete, including almost everything that is likely to be useful, the dictionary with broad categories that provides high recall and low precision should be used. If, on the other hand, only a few items are to be retrieved, but the user insists that these items must be relevant, then specific dictionary categories will be more useful.

In either case, problems will arise if words with very different frequency characteristics are included in the same category. Obviously, the effectiveness of the specific terms is much smaller if these terms are, in fact, considered equivalent to broader terms of high frequency. Thus each concept class should include only terms of roughly equal frequency so that the matching characteristics are approximately the same for each term within a category.

One note regarding the ambiguous terms is that they should be coded only for the senses that are likely to be present in the document collection to be treated. For example, the stem *field* must be shown in at least two separate categories – one corresponding to the notion of subject area, and the other to its technical sense in algebra; however, no category need be shown to cover the notion of a patch of land if the dictionary deals with the mathematical sciences or related technical fields.

2.4 Phrase Recognition

As observed earlier, concepts may occur in documents (or queries) as word groups or phrases. Thus, in attempting to perform subject analysis of stored documents or user search requests, we should locate phrases consisting of sets of words that are judged to be important in a given subject area. Various automatic phrase-recognition methods are possible – statistical approaches based on common occurrence of words in documents or approaches based on linguistic syntax or semantics. Statistical procedures are often unreliable because they lead to the identification of statistically meaningful but syntactically incorrect phrases. The approaches using syntactic and/or semantic analysis features have not met with much success because of the technical inadequacy and excessive cost.

More promising approach would be to take some human help. We use here a simple method, suggested originally in [43], based on the use of manually constructed *Phrase Dictionary* (PD). Before going into the details of this method, let us first look into the various difficulties involved:

- The concept denoted by a phrase is generally related to the concepts indicated by individual words and the same words may be combined in different ways to mean the same concept. For example, `language analysis`, and `linguistic analysis` mean the same thing. So the phrase recognition procedure should, in some sense, be insensitive to the surface syntactic structure of the phrase.
- Different words may be used to form phrases that mean the same concept. For example, the phrases `programming language` and `computer language` denote the same thing, whereas the individual concepts `programming` and `computer` are not the same.
- Phrases may contain any number of words.
- Constituent concepts of a phrase concept may not occur adjacently in a sentence but they may be spread.
- Sometimes the phrase components may be spread across sentences; and a given document or query may contain a concept without explicitly naming it. For example, a document dealing with computer science may not contain the phrase `computer science` at all. Since recognizing such phrases involves sophisticated semantical analysis, we will not attempt this here.

In light of the above observations, to view our goal in more formal and unambiguous terms, let us define a phrase as follows for all practical purposes of our method.

A phrase is a set of two or more distinct concepts (or word stems) not contained in the EWD, such that the components occur in the same sentence within a document or query.

With this definition we are limiting ourselves to recognize the phrase whenever the phrase components are present within the same sentence of a given document (or query). Note that we are not performing any checking on the co-occurrence of phrase components to make sure that permissible syntactic and semantic dependency relations exist. Thus any possible combination of the concepts `program` and `language` such as `programming language`, `languages` and `programs` and `linguistic programs` etc., would be recognized as a proper phrase `program language` which just denotes the co-occurrence of `program` and `language`. Another point to note is that our definition of phrases includes combinations of concepts rather than text words. These concepts are same as the ones that occur in SD.

Such a free interpretation of the phrase concept is justified since, for documents in a given area, presence of the individual components of a phrase in a given context generally

implies the presence of the complete phrase. Thus, for documents in computer science, the presence of the notions of program and of language normally implies that the document deals with the notion of programming language. As an example of this phenomenon, consider the sentence

For people in need of information, accurate retrieval is mandatory.

The phrase components information and retrieval are not syntactically related in the sentence; nevertheless, this sentence clearly relates to the topic area normally denoted by information retrieval.

Contrariwise, sentences for which the assignment of a given phrase concept would be clearly erroneous do not seem to occur in practice with sufficient frequency to cause any problem or they do not occur in those subject areas where any confusion might arise. Thus, consider a sentence such as -

People whose judgement is normally solid, state their convictions with great forcefulness.

This sentence does not deal with solid-state physics, even though the phrase components solid and state are both included; at the same time, the likelihood that such a sentence would occur in texts of solid-state physics is very small indeed.

Let us now summarize the phrase detection algorithm. Phrase dictionary provides the list of phrases of interest in the given subject area. Each entry gives a group of concepts that can co-occur to form a phrase. To identify the recognized phrase each entry will have a unique code consisting of a sequence of letters randomly chosen. To allow equivalence of phrases each entry may have multiple concept groups. Each sentence is subjected to the phrase recognition procedure which searches the sentence to recognize the concept groups that occur as defined in the PD. If all component concepts of a particular phrase occur in the sentence, then that phrase is assigned as an index term to the sentence.

As with other dictionaries, PD has to be constructed by the database manager apriori. No standard Phrase Dictionary can be supplied because it is by no means easy or practical to compile such an exhaustive dictionary for all possible phrases. However, given a certain subject area, it is possible by using published glossaries and a study of few typical documents in this area, to construct a PD that includes many of the phrases used to designate important concepts in that particular area. An automatically generated statistical data such as word frequency analyses and concept co-occurrence statistics may also be of help.

Stemming Algorithm

This chapter deals with the development of a stemming algorithm for Indian Languages. A stemming algorithm is a computational procedure to reduce all words with the same root to a common form.

Stemming algorithm is desirable at many places in the areas of Computational Linguistics and Information Science. Two obvious applications in Computational Linguistics are in the context of stylistic analysis of natural language texts and mechanical translation. In stylistic analysis, when evaluating the “richness” of a style, one is more interested in statistics on roots than on actual words; and any mechanical translation process will involve a process of stemming a word into its root and finding the equivalent root word in the target language and expanding (an inverse process of stemming) it into the corresponding word. In the field of information retrieval systems, it is useful in the content analysis of documents and query analysis. It also helps in compressing voluminous word lists or dictionaries maintained in a given system. In dealing with Indian Languages, compression is essential, as a full form dictionary can be many times the size of a root word dictionary.

In this chapter, we first look into the word formation process in Indian languages. We then briefly describe the suggested stemming algorithm and identify various theoretical and practical attributes of it. Each of the steps involved in the stemming algorithm are discussed in detail. We look into another application of the stemmer, namely, a spelling checker, that is of interest in the context of document retrieval, at the end of the chapter.

3.1 Words and Word Formation

A word is a unit of language that, in text, appears between spaces or between a space and a special symbol (like comma, hyphen, fullstop, exclamation etc.). Words are not the smallest units of meaning and syntax in a language. Instead, words are constructed of smaller parts, called *morphemes* by most linguists. Many simple words consist of only a single morpheme – some examples are *dog*, *text* and *book*. Other words are more complicated containing multiple morphemes – for example, *textbook* is a word containing two morphemes *text* and *book*. The identification, analysis, and description of morphemes, as well as the study of word formation, is called *morphology*.

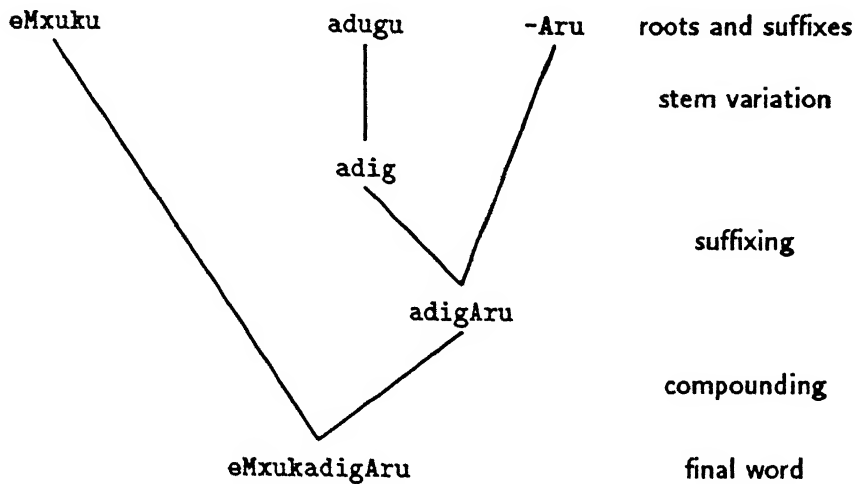


Figure 4: An Example of Word Formation in Telugu

Morphological units that serve as the basis of words are called *roots*. A root is a morphologically unanalyzable form from which stems and words may be derived via affixation. A *stem* is a morphological unit to which an *affix* attaches. An affix that attaches to the beginning of a stem is called a *prefix* and an affix that attaches to the end of a stem is called a *suffix*. For example, consider the English word *overburdened*; *over*, *burden* and *ed* are all morphemes. The morpheme *burden* is the root. The prefix *over* attaches to the stem *burden*, to result in *overburden*, to which the suffix *ed* attaches. In Telugu, prefixes are almost non-existent.

Stems may vary slightly in spelling when affixes are attached to them. The type of spelling change that occurs depends on both the stem and the affix. For example, the stem *entry* changes to *entrie* when the suffix *s* attaches to it, whereas the stem *book* doesn't. The stem *absorb* becomes *absorp* in *absorption*, while it does not change in spelling in *absorbing*. Handling the spelling changes is the most difficult step in any stemming algorithm.

Another common process through which words are combined in many languages is known as *compounding*. Compounding in English is limited to concatenation of two independent words. But in Indian Languages, *Sandhi* takes place during word compounding. *Sandhi* is a process where two phonetic elements coming into close contact are substituted by a third single phonetic unit. For example, with *Guna Sandhi* rule, a or A coming in contact with i, u and q will result in E, O and ar successively, as in *praXAna + upAXyAyudu = praXAnOpAXyAyudu*. A *Sandhi* type depends upon the last phonetic element of the first word and the first phonetic element of the second word. When a phonetical change occurs with *Sandhi*, the word spelling also changes.

Figure 4 illustrates the word formation process with an example from Telugu. The root *adugu* is changed to *adig* before the suffix *-Aru* to result in *adigAru*. The two words *eMxuku* and *adigAru* are combined using *Ukara Sandhi* (which says that when any vowel comes in contact to u at the word boundary, the u is lost) to form the compound word *eMxukadigAru*.

Compound word analysis is not included as part of our stemming algorithm for the following reasons.

- The component words of a compound, generally, cannot serve as good indexing terms. For example, it doesn't make sense to use the individual words `earth` and `quake` as indexing terms, instead of a single word `earthquake`. Same is the case with most of the words in Indian languages. On the other hand, leaving the compounds as they are improves the retrieval precision of the system.
- In a good number of cases, the meaning of a compound word has no direct relation with the meanings of individual words that make it up. Thus, to take examples from English, a `greenhouse` is not a house that is green; a `blackboard` is not necessarily black; and, `hot-dog` certainly doesn't normally mean a dog who is hot.
- Compound word analysis drastically slows down the stemming process. It is very easy for a language speaker to recognize a compound word and split it properly. For example, given the word `textbook`, it is easy for him to identify the constituent words `text` and `book`; this is because of his knowledge about the language – he knows that both `text` and `book` are valid English words. When this process has to be automated, the program has to do a lot of processing – it should derive all possible splittings and check which of them results in valid words. This later step again involves the stemming procedure and a dictionary look up.

In Indian languages, this process is even more complicated because the word compounding can take place not only by a simple juxtaposition of individual words, but also by Sandhi. A modern Telugu grammar book lists about 25 most commonly used Sandhi rules. This results in a large list of possible splittings for a given word, from which the correct one has to be selected by stemming and dictionary lookup.

Thus, the costlier operation of compound word analysis which doesn't seem to add to the system performance is not included in the implemented stemming procedure.

3.2 Stemming Algorithm

Stemming is the inverse process of word formation. Given a surface form word, the stemming algorithm has to generate the root (multiple roots in case of compound words) of the word. A natural solution to this, thus, involves reversing the sequence of steps of the word formation process. An additional step that is required to resolve various ambiguities resulting from the inherently ambiguous stemming process is a lexicon lookup.

Let us, first, briefly outline the suggested stemming algorithm. Given a word, the first step is to see whether the word occurs as an exceptional word. This check is performed by looking up a dictionary of exceptional words, which gives the root word for each exceptional word. If the given word is found in this, the corresponding root is returned; otherwise, the regular stemming process involving the following sequence of steps is tried.

The next step is to strip out the longest possible suffix from the word. A *suffix dictionary* provides the list of all valid suffixes; these suffixes are grouped into suffix categories

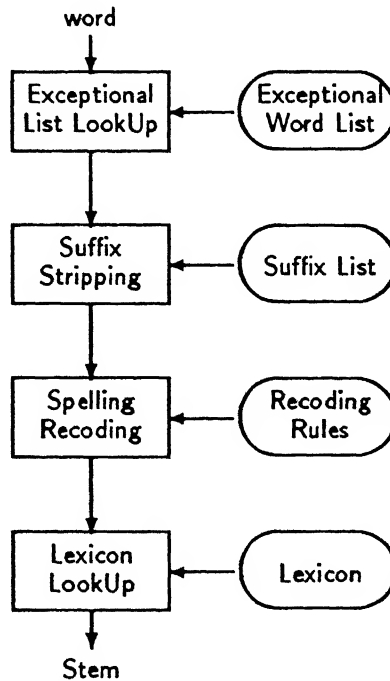


Figure 5: Steps Involved in the Stemming Algorithm

and each suffix category is given a category-code. The outcome of this step is the stem after cutting out the longest applicable suffix and the category-code of the suffix.

The next step involves performing a spelling transformation on the stem. This step is required to take care of certain morphological transformations a root word undergoes when a suffix attaches to it¹. This step, called *stem recoding*, uses a *recoding rules dictionary* which provides a list of transformations associated with each suffix category to recode stems resulted from the suffix stripping step². The stem recoding step may result in multiple possible roots from the stem. The right one from these is selected by looking up in the dictionary of root words or *lexicon*. In case, none of the candidate roots is found in the lexicon, the sequence of steps – suffix stripping, stem recoding, and lexicon lookup – is repeated with smaller suffixes until successful.

Figure 5 shows various steps involved in our stemming algorithm. A more detailed flow chart of the algorithm appears in Figure 6. Each of these steps are discussed in more detail below.

3.3 Suffix Truncation

Two main alternatives are possible for implementing the suffix stripping step – *iteration* and *longest-match* [31].

¹The reader is referred to Section 3.4 for examples of such morphological transformations in Telugu.

²Specific format used for the recoding rules dictionary and the actual procedures employed for recoding will be more clear in Chapter 4.

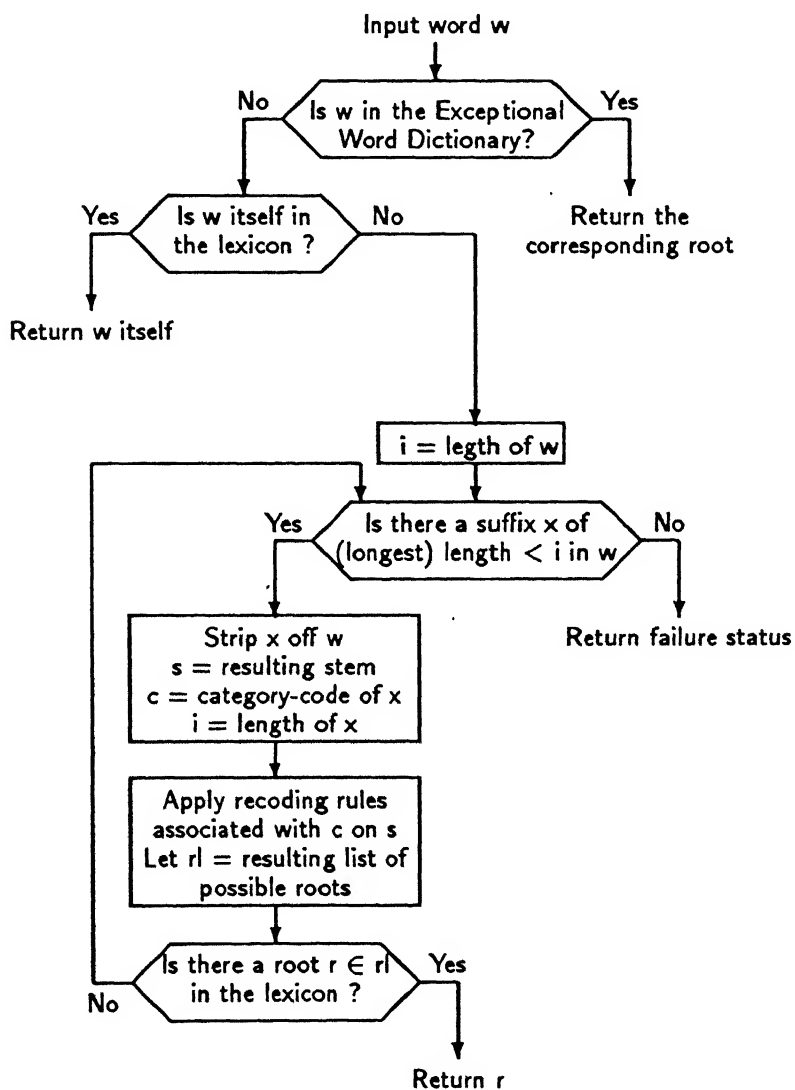


Figure 6: The Stemming Algorithm

Iteration is based on the fact that suffixes are attached to stems in a certain order, that is, there exist ordered-classes of suffixes. Each order-class may or may not be represented in a given word. For example, in Telugu, finite form verbs are formed by first attaching a tense suffix and then a personal suffix as in

```
winnAdu  =      win (verb stem)
                +  nA (past tense suffix)
                +  du (personal suffix of first-person singular)
```

An iterative stemming algorithm removes suffixes in each order-class one at a time, starting at the end of a word and working toward its beginning. No more than one match is allowed within a single order-class, by definition. One must decide how many order-classes there should be, which endings should occur in each, and whether or not the members of each class should be internally ordered for scanning.

With a simple longest-match stemming algorithm, only one order-class of suffixes is used. All possible combinations of suffixes are compiled and then ordered on length. If a match is not found on longer endings, shorter ones are tried. For example, the verb suffixes *naxi* and *daxi* precede *axi* which precedes *i* in the list.

One obvious disadvantage of the longest-match method is that it requires generating all possible combinations of suffixes. This disadvantage is also present to a large degree with iterative algorithm with as many order-classes as possible. To set up the order-classes, one must examine a great many endings. Furthermore, it is not always obvious to which class a given suffix should belong. It is also possible that the occurrence of members of some class is context dependent. In short, while an iterative algorithm requires a shorter list of endings (all order-classes together), it introduces a number of complications into the preparation of the list and programming of the routine.

A second disadvantage of the longest-match method could be the amount of storage space the endings require compared to an iteration algorithm. However, it is not an immediate problem because a one-class list of endings for Telugu (in fact, for any Indian Language) may not have more than a thousand entries.

So, it was decided to use a longest-match (i.e., noniterative) stemming algorithm with one-class list of endings. That is, the intuitively inefficient procedure that requires listing all possible suffix combinations has been followed in order to minimize lot of complexities that are going to arise in implementing and preparing ordered-classes of suffixes for a not-well-understood iterative procedure.

3.4 Stem Recoding

The stem recoding step handles “spelling exceptions”, mostly instances in which the same root varies slightly in spelling according to what suffixes originally followed it. The examples, from Telugu, given in Figure 7, show some of the range and type of variations that may occur. The stem is separated from the ending by a vertical bar.

adugu	adugu wU	adig Exi	adag akuMda
pilucu	pilus wU	pilic Exi	pilux xAM pilis wE pilav akuMda piluv u
wIyu	wIs wU	wIx xAM	wIy aMdi wI kuMda wiyy aMdi
kuxurcu	kuxurus wU	kuxiris wE	kuxurux xAM kuxirc Adu kuxarc akuMda kuxurc u
kottu	kodu wU	kott Adu	
vaccu	vas wAdu	vax xAM	vacc Adu rA kuMda r aMdi

Figure 7: Examples Showing the Need for Stem Recoding

adugu	adugu	adig	adag
pilucu	pilus	pilic	pilux pilis pilav piluv
wIyu	wIs	wIx	wIy wI wiyy
kuxurcu	kuxurus	kuxiris	kuxurux kuxirc kuxarc kuxurc
kottu	kodu	kott	
vaccu	vas	vax	vacc rA r

Figure 8: Possibility of Recoding by Listing out Equivalent Stems

While such spelling changes do occur only before certain suffixes, this set of suffixes is usually quite large. Thus, it is not practical to consider the exceptional stem endings as part of the suffixes in a one-class suffix truncation algorithm such as the one we are using; the number of extra suffixes that must be included to do so is prohibitive³.

One simplest method that comes immediately to mind is to construct a list of equivalent stems. For example, to deal with the words shown in Figure 7, we have to provide a list of equivalent stems as shown in Figure 8. This solution is not practical because of the size of such list to be provided – which leads to difficulties in preparing it; also, this technique requires large amount of memory and processing time is higher because of the greater number of disc accesses.

³Such a simple strategy is followed, for example, in the morphological analysis algorithm suggested in [6].

Two different stem recoding procedures are implemented in this work to take care of spelling exceptions.

Rule-based Recoding With this approach, a stem is recoded based on a sequence of context-sensitive transformational rules. The context-sensitiveness of these rules lies in the fact that certain transformations are applied only when certain suffixes are cut. For this purpose, suffixes are grouped into different categories and each category is given a category-code. The suffix truncation step outputs not only the stem, but also the category-code of the suffix that is stripped off the word to result in that stem. Each category-code is associated with a set of recoding rules. The recoding step applies only the recoding rules associated with the given category-code. The details of specific format used for recoding rules and particular procedures employed for applying them are provided in Chapter 4.

It is important to note that the rules used in recoding should be not only context-sensitive but also *ordered*. Consider the following three rules (refer to Appendix B):

1. In disyllabic stems of the type (C)VCa, where C stands for consonant and V stands for vowel, replace final a by u.
2. Delete stem final u if any.
3. Replace stem final d and b by tt and pp respectively.

Now consider the stems *ceba*, *cepp* and *cebu*. If the rules are applied in the order given, all these stems will map to *cepp*; if they are applied in a different order, say 3, 2 and 1, the results would be,

$$\begin{array}{l} \text{ceba} \xrightarrow{3} \text{ceba} \xrightarrow{2} \text{ceba} \xrightarrow{1} \text{cebu} \\ \text{cepp} \xrightarrow{3} \text{cepp} \xrightarrow{2} \text{cepp} \xrightarrow{1} \text{cepp} \\ \text{cebu} \xrightarrow{3} \text{cebu} \xrightarrow{2} \text{ceb} \xrightarrow{1} \text{ceb} \end{array}$$

which are incorrect.

The transformational rules can be directly coded into the program i.e., a set of procedures can be written to simulate these rules. But such an approach is not flexible enough – it has to be used for a different language, one has to write again a new set of procedures; and even for the same language, we cannot easily add a new rule into the recoding step. So we must write an interpreter like program to which we can input the recoding rules in a prespecified standard format for a given language; it then applies these rules on stems to allow the ultimate matching of varying stems.

Recoding by Stem Ending Replacement This technique is, methodologically, not very different from the rule-based recoding. Stem recoding is done by simply replacing a particular sequence of characters at the end of the stem by another sequence of characters,

based on the category-code of the truncated suffix. Each suffix category is associated with a set of stem ending replacement rules. These rules are tried in the decreasing order of the size of ending to be replaced. Generally, the first applicable ending replacement rule turns out to be the right one. When there are multiple replacements applicable, the lexicon lookup step, anyway, resolves the ambiguity.

A crucial difference between rule-based recoding and recoding by stem ending replacement may be noted. Rule-based recoding involves applying a sequence of rules; at the end, the result is a list of possible roots, the correct one from which is selected by lexicon lookup. Whereas, in recoding by stem ending replacement, each rule is complete by its own; at the end of application of each rule, lexicon is looked up to see if this results in the correct root. Otherwise, that rule is completely thrown away and next rule is tried. In other words, the former involves a bit of iteration, whereas, the latter is completely a longest-match-first approach.

Instead of asking the language expert to supply the rules for recoding by stem ending replacement, the program itself generates these rules based on a set of *paradigm tables* provided by the language expert. The specific format used for paradigm tables and the procedures employed for deriving stem ending replacement rules from these are described in Chapter 4.

3.5 Lexicon

One component of the stemming program that one prefers to avoid is the lexicon⁴. This has turned out to be unavoidable for Indian languages because of their complex morphological structure which cannot be governed by simple suffix stripping and spelling transformational rules⁵. The following paragraphs illustrate the importance of lexicon.

Difficulties in Suffix Truncation There are lot of complications involved in the application of suffix stripping process. Suffixes introduced for one class of words trouble the proper stemming of some other words. An example is the matching of the suffix -Adu to the word Adu to result in a null stem, to the word pAdu to result in a single letter stem, or to the word pOrAdu to result in an ambiguous stem, all being improper matches; this suffix is actually meant for verbs in past-tense third-person masculine singular form. Such improper stripping can be avoided to a great extent if we have some syntactic information about the word utterance. Unfortunately, this is not possible.

We can approach this problem by having a restriction on the resulting stem length when a suffix is stripped. By restricting that the resulting stem must have atleast two letters, we can avoid the first two improper truncations but the third is still not solved because increasing this limit on minimum stem length will trouble other proper truncations.

⁴However, the use of lexicon is not uncommon in morphological analysis programs for information retrieval [22].

⁵The lexicon becomes compulsory when compound word analysis is employed

Other possibility is to place some qualitative contextual restrictions like, “do not remove this suffix when a specified pattern of letters is found in the resulting stem”. But such generalized rules are extremely difficult to draw for languages like Telugu; every rule one tries to arrive, he'll find many exceptions to it.

If a lexicon is available, such improper truncation will not matter, because when we find that when a particular suffix is stripped, the resulting stem is not in the lexicon, we can try the remaining suffixes.

Problems in Recoding Similar type of complexities exist in the recoding process also. In proposing a recoding procedure, we made the assumption that most of the spelling variations that can occur can be adequately covered by a small set of context-sensitive transformational rules. Though Krishnamurthi[28] shows that the process of word formation from roots in Telugu can be easily described by a general *item and process* model unambiguously, the reverse process of stemming cannot be. Consider, for example, a rule which says[28],

A consonant pair C_1C_2 such that $C_1 = C_2$ or C_1 is nasal and C_2 is homorganic stop or affricate is simplified to C_2 at stem boundary when a suffix starting with a consonant is attached.

Applying this rule, *vacc* becomes *vac*, *ceppiMc* becomes *ceppic*, *veLL* becomes *veL* and *wann* becomes *wan*. But how do we apply the reverse rule. We are given the stem *ceppic*, how do we know whether it should be transformed to *ceppiMc* or *ceppicc*? Apart from this inherent ambiguity in the recoding rules, many of the rules also have lot of exceptions which are not always predictable and there are great many stems whose behaviour cannot be governed by such rules.

This problem could be solved to a great extent by providing a lexicon. When an ambiguous rule is used to perform stem recoding, the correct root from the resulting set of candidate roots can be selected by looking up in the lexicon. There may be cases, where two or more of the candidate roots are in the lexicon. Such cases, however, are too rare, and no simple way to handle such cases seems to exist.

New Words Lot of new technical words are getting introduced into Telugu now-a-days. Suppose a new word appears in a document whose morphological characteristics are unknown at the time of preparing the suffix tables and recoding rules; the system may fail to handle it properly. When a lexicon exists, the system identifies that this is a new word and database maintainer may accordingly make entries for it in SD, PD and/or the exceptional word list for the stemmer.

Multiple Spellings Some words in Telugu have multiple spellings. Such cases cause problem because if one spelling is used in documents and the user request contains a different spelling for the same word, the retrieval operation fails. Existence of a lexicon atleast identifies new spellings and informs the user and the database manager of this. Some example words with multiple spelling:

- Foreign words could have different spellings –

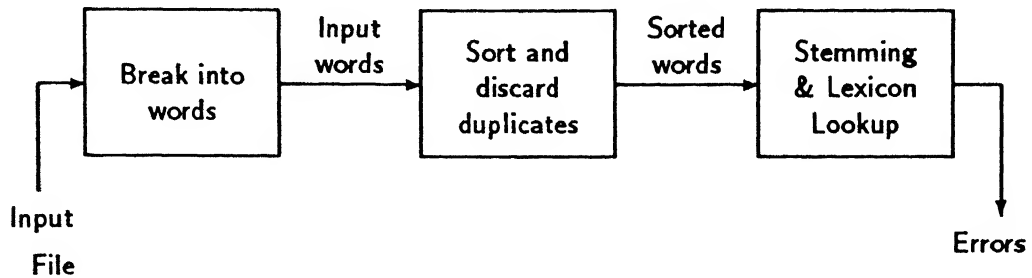


Figure 9: Spelling Checker

basu/bassu/bas (bus)
 ProMtU/PraMtU (front)
 pennu/pen/pEna (pen)
 pAMtu/pEMtu/pyAMtu (trousers)
 sUtKEs/sUtKEsu/sUtukEsu/sUtikEsu (suitcase)
 kyAraMbOrdu/kEraMbOrdu/ kyAraMbOrd/kEraMbOrd (caram board)

- Certain native stems also have multiple correct spellings –

puvvu/pUvu (a flower)
 veyyi/vEyi (thousand)
 caMxrudu/caMxurudu (the moon)

- Proper names when written may have different spellings –

kqRNudu/kriRnudu
 heYxarAbAx/heYxrAbAx/heYxarAbAdu/heYxrAbaxu/heYdrabad

Misspelled words Misspelled words may appear because of author ignorance or keypunch errors. Such misspelled words may occur in both documents or queries. A spelling checker program would help the document authors recognize any misspellings. The heart of any spelling checking program lies in the lexicon.

In our framework, lexicon provides a list of all proper roots. No additional syntactic or semantic information about roots is incorporated into the lexicon. The implementation details of lexicon lookup are discussed in Chapter 4.

3.6 Spelling Checking

Spelling errors can be introduced into documents in many ways:

Author Ignorance Such errors can lead to consistent misspellings and are probably related to the difference between how a word sounds and is actually spelled.

Typing Errors These are less consistent but perhaps more predictable, since they are related to the position of keys on the keyboard and probably result from errors in finger movements. Studies have shown that large document stores or databases may have significant keypunching errors [7].

Transliteration Errors With the current method of inputting Indian Language documents using Roman keyboard, errors due to ignorance of the used transliteration scheme are possible to occur.

Errors with OCR When the Optical Character Recognition comes to a successful end, it is hoped that it is used for automatically scanning published documents and storing in the system. Errors may occur in character recognition.

Spelling errors can occur in user queries also in similar fashion. Such misspellings which are tolerable in normal everyday correspondence, with computerized systems like automatic information retrieval create lot of problems. Many earlier experimental studies concluded that spelling errors are severe enough to degrade the system retrieval performance substantially[7, 14].

Stemming algorithm described above could be easily used to detect misspellings. Idea is to report a misspelling whenever the stemming process fails to obtain the constituent root(s) of a given word. However, the algorithm may fail to report some misspellings though it doesn't report a correct word as a misspelled as long as the stemming rules and lexicon are complete enough. Such incorrect reporting is possible due to two reasons ⁶ –

1. The stemming algorithm may incorrectly map a misspelled word to a correct root word in the lexicon.
2. Our implementation of lexicon based on probabilistic hashing technique may accept an incorrect stem as correct one due to hashing collisions.

However, such mistake by the spelling checker is too rare to question its usefulness. With the current implementation, the stemmer does not perform compound word analysis; hence, the spelling checker also fails for compound word.

Spelling checking of query words is straight-forward – each misspelled word is reported on-line to the user. For spelling checking of documents, a batch program is written which consists of the following sequence of steps, as shown in the Figure 9:

1. Split out the words of the document.
2. Cull the words for duplicates by sorting them.
3. Perform stemming and lexicon lookup and accept all words that mapped to lexicon entries.
4. Report all remaining words as potential spelling errors.

This approach is similar to the one used in Unix Spelling Checker *spell*[5, 33].

⁶Note that errors like improper suffixing and illegal word compounding may also go undetected like with most of the modern spelling checkers. However, these are not the “type” of spelling errors we are aiming to correct.

Implementation

This chapter deals with the implementation of the content analysis procedure for Indian languages. We present an overview of various issues and difficulties involved in implementing the information structures and features that were identified to be potentially useful as part of an information retrieval system for Indian languages in the earlier chapters. A more detailed description of some of the important modules used in the implementation can be found in Appendix C. The source code is available in public domain. More details can be obtained from that.

The most important and complex part of the implemented content analysis scheme is the stemming algorithm. Hence, most part of this chapter is devoted to the details of implementation of the stemming algorithm. At the end of the chapter we look into the other components.

4.1 Stemming

As we discussed earlier, our approach to stemming involves a four step process – exceptional word lookup, suffix truncation, recoding and lexicon lookup. At least, the second and third steps can be hard-coded using either a series of lookup tables or a set of switch or if-else statements. The hard-coded approach has its advantages – hard-coded analyzers tend to be very efficient and fast – which is really important in user-oriented interactive information retrieval system. But at the same time, they are difficult to maintain. When we are developing a stemmer for a new language, its handy to be able to add new suffixes, new rules and new stems to the stemmer or delete some without too much work.

With this motivation, the developed program for stemming is kept general. It is totally input-driven. It accepts an exceptional word dictionary, a suffix dictionary, a recoding rule dictionary, and a lexicon of valid root words, and performs stemming on input words. To use this stemmer for other languages, the only thing one has to do is to supply these dictionaries. The fact that such a generalized analyzer tends to be typically little slower than a hard-coded one, is a small price payed for its generality and easy-extendibility. Once the language is stable, i.e., the stemming rules are completely developed and tested, one can always go back and hard-code a stemmer, if it's really necessary to speed it up.

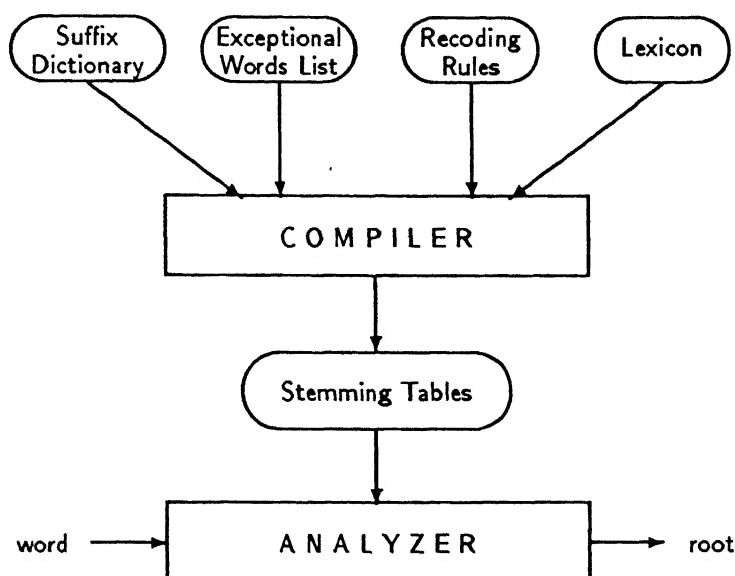


Figure 10: The Compiler and Analyzer components of the stemmer

To speed up the stemming process, our stemming tool has two components : a *compiler* and an *analyzer*. The compiler component takes various dictionaries in the format provided by the linguist or the system manager, and generates a set of tables and data structures from these. These tables are then used by the analyzer component to perform the actual stemming. Figure 10 shows how these two components interact.

The remainder of this section presents important details of these two components of the developed stemming tool, along with the underlying theory wherever necessary. The details of lexicon implementation are postponed to the next section.

4.1.1 Suffix Dictionary

The suffix dictionary provides a list of all valid word suffixes grouped into different categories. All suffixes belonging to a particular category behave similarly in terms of the way stems change their spelling when these suffixes are attached. Each category is identified by a category-code, basically a positive integer. The recoding step uses the suffix category-code as an index into the applicable recoding rules.

The specific format used for suffix dictionary file is as follows. It contains a list of entries, where each entry is a category-code followed by suffixes belonging to that category. Single entry may spread across lines; but each entry should start on a different line. Comments can occur any where on any line and they start with a %; the part of the line starting with a % is neglected. The reader is referred to Appendix B for examples that provide a more clear picture of this format.

The compiler component of the stemming tool generates a trie of reverse suffixes from the suffix dictionary. This data structure allows a rapid matching of word suffixes[26, 18]. Each node of the trie is represented by a vector whose subscripts run from 0 to M , where

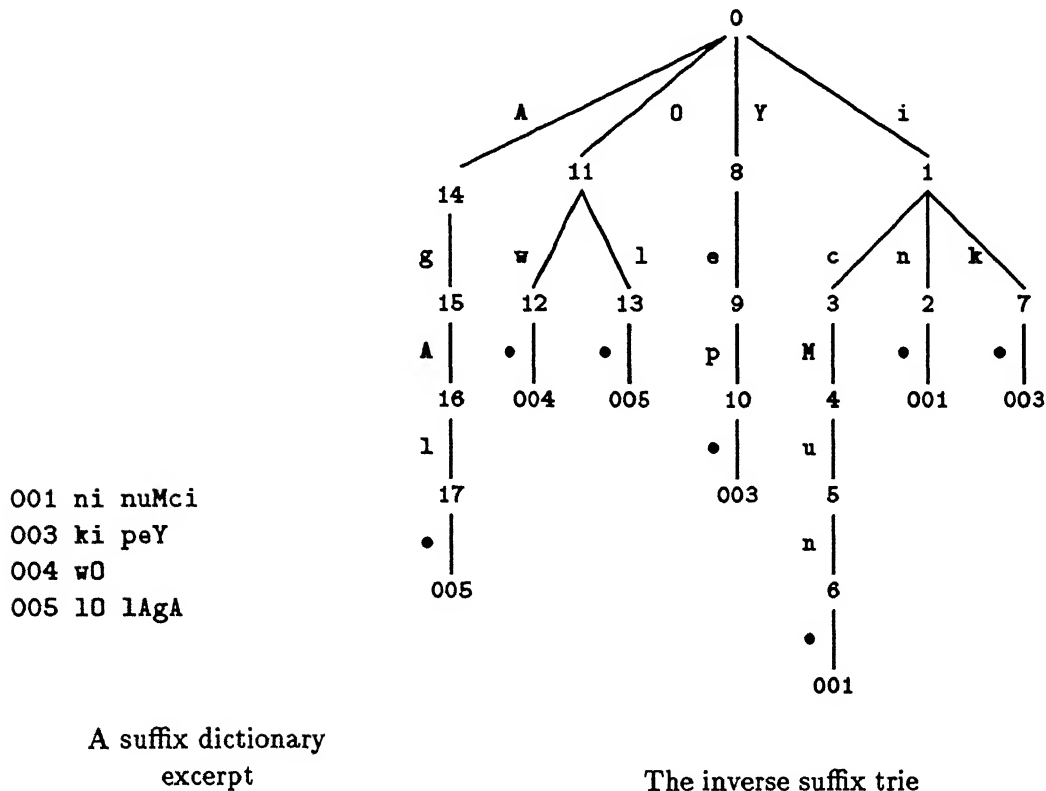


Figure 11: An example of inverse suffix trie

M is the size of the alphabet¹. The 0th element in this vector is used only for “external” nodes and contains the category-code of the suffix to which it corresponds. Other vector elements are pointers to child nodes within the trie. Example given in Figure 11 makes these points more clear.

The trie is then written to a file, which will be read later by the analyzer component. The analyzer component uses the trie for detecting the longest valid suffix of a given word. The specific trie insertion and lookup algorithms used are given in Appendix C.

4.1.2 Rule-based Recoding

The stem recoding step involves altering the stem resulted after a suffix has been cut off, to get the root word. In rule-based recoding, this step is performed based on an ordered set of context-sensitive transformational rules. These rules are provided by a language expert and are expressed in a regular expression like language.

The compiler component constructs a set of state machines which simulate these rules. It prints tables describing these state machines on a separate file. The analyzer reads this

¹In our case, M is 52. Note that some of the letters, e.g., z and Z, are not used in the encoding scheme given in Appendix A. Even then, M is kept 52 to keep the implementation independent of the encoding scheme.

machine	→ rule_group machine
	rule_group
rule_group	→ rule rule_group
	rule EORG
rule	→ pattern : action : suf_cat_codes
pattern	→ [^] [expr] { [expr] } [expr] [\$]
expr	→ expr cat_expr
	cat_expr
cat_expr	→ cat_expr term
	term
term	→ [expr]
	(expr)
	character
action	→ repl : number , number
repl	→ string , repl
	string
suf_cat_codes	→ number , suff_cat_codes
	number
character	→ a ... z A ... Z 0 #

Figure 12: Grammar of Recoding Rules Specification

file and runs the state machines to perform stem recoding. In the following paragraphs, we describe the format used for expressing rules; we also provide a brief description of the procedures used for constructing and running the state machines. These procedures are given, in more detail, in Appendix C.

Rules Specification

The grammar used to recognize a recoding rules specification is summarized in Figure 12.

Each rule is specified on a single line and has three parts separated by colons – a pattern part, an action part and a suffix category-codes part. The action (essentially a string replacement) as specified in the action part is performed on a given stem only when the specified pattern is found in the stem and the suffix category-code matches one of the category-codes specified in the category-codes part. Spaces can freely occur anywhere in a rule except within the pattern part.

Rules are grouped into rule-groups and the order of rule-groups is important as noted in the previous chapter. However, within a group, the order of rules is not important and it is assumed that in any given case atmost one of the rules in a particular rule-group are applicable. Rule-groups are separated by a line starting with a star (“*”) in the input

file. This line can be used as a comment line or can be left as a blank line except for the first character.

A pattern has the following shape:

`^expr1{expr2}expr3$`

Each component of this is optional except { and }.

- The uparrow anchors the pattern to the start of the word. For example, the pattern `^an` matches the `an` in `and`, but not the `an` in `hang`.
- The dollar sign anchors the pattern to the end of the word. The pattern `an$` matches the `an` in `fan`, but not the one in `and` or `hang`.
- `expr2` actually matches the part of the word that is subjected to replacement whereas `expr1` and `expr3` determine the context. For example, the pattern `{a}` matches any `a` in `character` but the pattern `h{a}` matches only the first `a` and the pattern `{a}ct` matches only the second `a`.

Each expression is just a series of letters (`a..z` and `A..Z`) and metacharacters (`([] () | * @ < >`). A simplest expression is just a series of letters like `and` which matches the sequence of same letters in the input. The metacharacters can be used to describe more complex strings –

- The vertical bar `|` is used for alteration. Two expressions separated by a `|` recognize a match of the first expression OR a match of the second. The expression `either|or` matches `either` or `or`.
- Brackets `[]` are used for grouping. The expression `[b|m]at` matches both `bat` and `mat`.
- Parentheses are used to show optional parts. The expression `ba(na)na` matches two strings : `bana` and `banana`.

In addition, the metacharacters `<` and `>` are used to provide a simple macro support. Macro definitions are listed at the beginning of the file as simple name-text pairs. These macros, then, can be used in an expression by writing the macro name enclosed in `<` and `>`. Macro expansion is done by simply replacing a `<name>` by the corresponding text. Two wildcard characters `*` and `@` are used to provide implicit macros. A `*` matches any letter representing a consonent in the language of interest and a `@` matches an vowel. Using this two characters than writing otherwise using macro facility or not reduces the number of states in the resulting state-machine.

State Machine

Finite state automaton or *finite state machine* is a well known conceptual tool used for recognizing strings matching a given pattern. Ofcourse, we need to do much more than pattern recognition which is achieved by adding some sort of an extra layer to a simple state-machine. Strictly speaking, an FSM consists of the following:

- A finite set of states.
- A finite set of input alphabet.
- A set of transitions from one state to another. Each transition is labeled with a character from the input alphabet.
- A special start state.
- A set of final or accepting states.

A *deterministic finite automaton* (or DFA) is a finite state machine in which each transition has a non- ϵ label and no two transitions from a given state have the same label. More general type of state machine is a *nondeterministic finite automaton* (or NFA). An NFA has no limitations on the number and type of transitions. Two outgoing transitions can have the same label, and the transitions can be labeled with a ϵ . An ϵ -transition matches an empty string and hence is taken without advancing the input and is always taken – regardless of the input character.

As one can see, an NFA can be an awkward data structure to use. Theoretically an NFA often has fewer states than an equivalent DFA because it can have more transitions from a single state than a DFA has. Nonetheless, this sort of NFA is difficult to construct and represent in a computer program. For our purposes of comparison, NFA can have many more states than an equivalent DFA, and it is difficult to write a driver (i.e., the analyzer part) that can use it directly.

These problems are solved by creating state machine in a two step process. First, an NFA representing the input rules is constructed which is then converted into a DFA.

Using State Machines for Recoding

As mentioned above, recoding process is governed by a sequence of groups of rules, with the order of rules within a group unimportant. State machines are used for performing recoding based on these rules as follows. One DFA is used to simulate each rule group; and all the DFAs are connected to form a sequential chain. The recoding process involves running these machines, in sequence, on the given stem. For a single stem, a machine can result in multiple possible stems (this depends totally on the rules specification). Hence, at the end, a set of possible roots for the given stem are generated. A lexicon lookup later selects the correct root.

Constructing State Machines

To build the chain of state machines, the compiler part first constructs NFAs from the input rule specifications using a scheme similar to *Thompson's construction*[3]. A machine built with such construction has several useful characteristics :

- All the machines that recognize individual patterns – no matter how complicated – have a single start state and a single end state.

- No state has more than two outgoing transitions.
- There are only three possibilities for the labels on the transitions :
 1. there is only one outgoing transition labeled with a single input character
 2. there is only one outgoing transition labeled with ϵ
 3. there are two outgoing transitions labeled with ϵ .

There are never two outgoing transitions labeled with input characters, and there are never two transitions, one of which is labeled with an input character and the other with ϵ .

The following data structure uses these characteristics to implement an NFA state.

```
/* Structure definition for an NFA state
 */
typedef struct nfa
{
    int      edge;      /* lable for edge; character, EMPTY or EPSILON */
    struct nfa * next;   /* next state (or NULL if none) */
    struct nfa * next2; /* another next state if edge == EPSILON */
    struct nfa * rgedge; /* points to starting state of next rule-group */
    char *     replace; /* NULL if not accepting state, else a pointer */
                                /* to the replacement string */
    int      strtPos; /* strtPos and endPos together determine the */
    int      endPos;  /* part of the stem to be replaced; included */
                                /* to simplify the program */
    int *     catCodes; /* category-codes of suffixes to which the */
                                /* rule in this accepting state is applicable */
} nfa_t;
```

The next field either points at the next state or is set to NULL if there are no outgoing edges. The next2 field is used only for states with two outgoing ϵ edges. The edge field holds one of three values that determine what the lable looks like:

- If there is a single outgoing edge labeled with an input character, edge holds that character.
- If the state has an outgoing ϵ edge, then edge holds EPSILON.
- If the state has no outgoing transitions, edge is set to EMPTY.

The series of rgedge pointers put the machines simulating individual rule-groups together into a chain. These edges connect the starting states of individual machines.

The remaining fields are used only for accepting states and they contain values indicating “not applicable” for nonaccepting states. The three fields replace, strtPos and endPos provide the string replacement to be done when that accepting state is reached.

And the `catCodes` field provides a list of suffix category-codes to which the original input rule is applicable.

Once NFAs are constructed, the next step is to turn them into DFAs. The method used here is called *subset construction*[3]. The basic strategy is to make use of the fact that all NFA states connected with ϵ -edges are effectively the same state.

The following data structure is used for representing the DFA accepting states –

```
/* structure for storing information on accepting states of DFAs
 * Values for all the four fields are copied from the corresponding
 * NFA states directly */
typedef struct accept
{
    char *      replace;
    int         strtPos, endPos;
    int *       catCodes;
} accept_t;
```

All the DFAs are represented together using three arrays:

DTrans – the DFA transition matrix, an array as wide as the input character set and as deep as the maximum number of states. `DTrans[current-state][input-character]` gives next-state. One additional column is used to indicate whether a particular state is a final (accepting) state; if so, it contains pointer to the corresponding entry in **AStates**.

AStates – array of type `accept_t`; contains information of the accepting states. For each accepting state it stores the information that is required for changing the stem (recoding) when that state is reached, in running the machine.

RStart – As pointed out above, we are storing all the DFAs corresponding to different rule-groups in the same arrays (i.e., **AStates** and **DTrans**). The array **RStart** points to the starting states of individual machines in **DTrans**.

The compiler component of the stemmer, then, prints these three tables describing the machines to perform recoding in a file, so that these can be read later by the analyzer component.

Running the State Machines

The analyzer component first reads the file written out by the compiler component. The three arrays **DTrans**, **AStates**, and **RStart** are kept in the main memory during the stem recoding process.

Given a stem to recode, the state machines, each representing one group of rules, are run in sequence on the stem. Running one machine on a stem may result in multiple stems, because the rules themselves may indicate multiple possible string replacements. The following state machines in the chain are run on each of these stems. Running a state

machine is done in a *non-greedy* way, stopping once an accepting state is reached, at which point the string replacement as specified with that state is performed.

At the end of running all the state machines on the given stem, the result is a set of possible roots of the original word.

4.1.3 Recoding by Stem Ending Replacement

In this approach, stem recoding is done by simply replacing a sequence of characters at the end of the stem by another sequence of characters. The rules for this string replacement are derived from *paradigm tables* supplied by the language expert. Root words are classified into different paradigm classes based the way the endings undergo spelling changes when particular suffixes are attached. Only one representative from each paradigm class is included in the paradigm tables file.

Each entry in the paradigm tables file has the following format:

```
# padu
pad      : 052, 070
pada     : 060, 062
padu     : 060, 062
```

The line starting with a # specifies a root word and each of the following lines specify the form it takes when suffixes belonging to a particular set of suffix categories attach to it. Thus, the above shown entry indicates that the root padu takes the form pad when a suffix belonging to either of the categories 052 or 070 attaches to it; and the form pada when a suffix belonging to one of the categories 060 or 062 attaches to it etc.

The compiler component generates a set of ending replacement rules for recoding from the paradigm tables file. The data structure used for a rule is as follows :

```
/* structure definition for a stem recoding rule based on ending
 * replacement. */
typedef struct rcrule
{
    ushort   catCode;           /* category code of suffixes */
    char     *oldEnd;           /* end to be stripped off */
    char     *newEnd;           /* new end to be appended */
    ushort   minRootSize;      /* minimum stem size for this */
                                /* rule to be applicable */
} rcrule_t;
```

Generating these rules is a simple and straight-forward process. Let us illustrate this by an example. The above shown paradigm table says that the root padu takes the form pada when suffixes of categories 060 or 062 are attached to it. So, in the recoding step, when the stem pada has to be recoded, given that the stripped off suffix belongs to the suffix category with category-code 062, we must get back padu. This can be done by replacing

the ending a by u. So, the resulting replacement rule is : when the suffix category-code is 062 replace stem ending a by u, subjected to the condition that the stem has minimum 2 syllables. The syllable count is approximated by counting the number of vowels.

Thus the paradigm tables file entry shown above results in the following list of rules :

catCode	oldEnd	newEnd	minRootSize
052	NULL	u	2
060	NULL	NULL	2
060	a	u	2
062	NULL	NULL	2
062	a	u	2
070	NULL	u	2

A rule with catCode c, oldEnd o, newEnd n and minRootSize m is interpreted as follows: The rule can be used to recode a stem only when the suffix that was cut off belongs to the suffix category with category-code c and it applies only if the stem contains at least n syllables. To apply the rule, the ending of the stem, if it matches with o is replaced by n. A NULL value for the field oldEnd means that no ending is cut, but the newEnd is simply attached; and a NULL value for newEnd means that oldEnd is simply cut and nothing is attached back to the stem. However, when both oldEnd and newEnd are NULL, it means that the stem is left unchanged.

The compiler phase, in addition, does a little more processing to eliminate duplicate and redundant entries. The following steps are performed for this purpose :

- If a rule has both the oldEnd and newEnd fields equal, that rule is eliminated. This case is possible only when oldEnd = newEnd = NULL. Such a rule means that the root word does not change in its stem form. An additional step, which involves looking up the lexicon immediately after a suffix is deleted, is introduced into the stemming algorithm to take care of all such rules.
- If two entries have all the fields same, only one of them is preserved.
- If two or more rules have all the fields same, except for minRootSize, only the one with the least value of minRootSize is preserved.

These rules are then sorted on the increasing order of the field catCode and written onto a file. This file is read later by the analyzer part to perform stemming.

4.1.4 Exceptional Words List

An exceptional words list is required to handle all the cases when the regular stemming process fails to obtain the root of a word. Each line in the input file contains a root and its

all possible surface word forms. As with other files, comment and blank lines may appear anywhere in the file. The input exceptional words file is compiled into an array `excepList` whose entries have the following structure:

```
/* structure definition for the stemming exceptional list entry */
typedef struct excep
{
    char *    word;    /* a word */
    char *    root;    /* and its root */
} excep_t;
```

The array itself is sorted in alphabetical order on the field `word` so that a simple binary search is possible.

4.2 Lexicon

In this section, we look into the implementation details of the lexicon. In its simple abstraction, the process of lexicon lookup involves testing set membership. The requirement that it should be space-efficient and fast makes the problem complex. Literature suggests many possible data structures for implementing a lexicon [5, 15, 33, 37, 38, 41]. The best approach for our application appears to be a probabilistic hashing technique similar to the one used in Unix spelling checker program `spell`[5, 33].

Essentially, the technique involves replacing each lexicon entry by a hashed version of it. This hashed version is referred to as the *check hash*. The check hash can be significantly shorter than the average lexicon entry, thereby reducing storage requirements. The length of the check hash is fixed, which simplifies the dictionary structure and the program used to access it. Unfortunately, it is possible that two words will produce the same check hash. Such possibility of hash collision rules out the use of this technique in some applications; whereas in our application some occasional collisions may be acceptable. A check hash collision during lexicon creation will result in no errors. But during its use, a collision between a misspelled word and a lexicon entry may cause the misspelled word to be accepted. This is clearly undesirable, but provided that the frequency of errors is less, the overall quality of the system will not be significantly affected.

Suppose that we wish to test membership in a lexicon of v words and the algorithm uses b bit long hash code, then the possibility of false acceptance will be $\frac{v}{2^b}$. Assuming that we typically have a vocabulary of 50,000 words and accepted error rate once in 5000 words, we need a 28 bit long hash code. To make the program simpler, we select a multiple of 8 bits hash code, that is 32 bits. With this choice now the error rate would be once in a lack words and the required memory is just $50,000 * 4$ bytes or 200 Kbytes which allows us to keep the full lexicon in the main memory itself. This small lexicon can be read from disk quickly.

Let us now summarize how the lexicon lookup works. Initially the list of words L is given. The compiler component of the stemmer calculates the check hash value $h(w)$ for every $w \in L$, sorts the resulting list of check hash values and writes onto a secondary file.

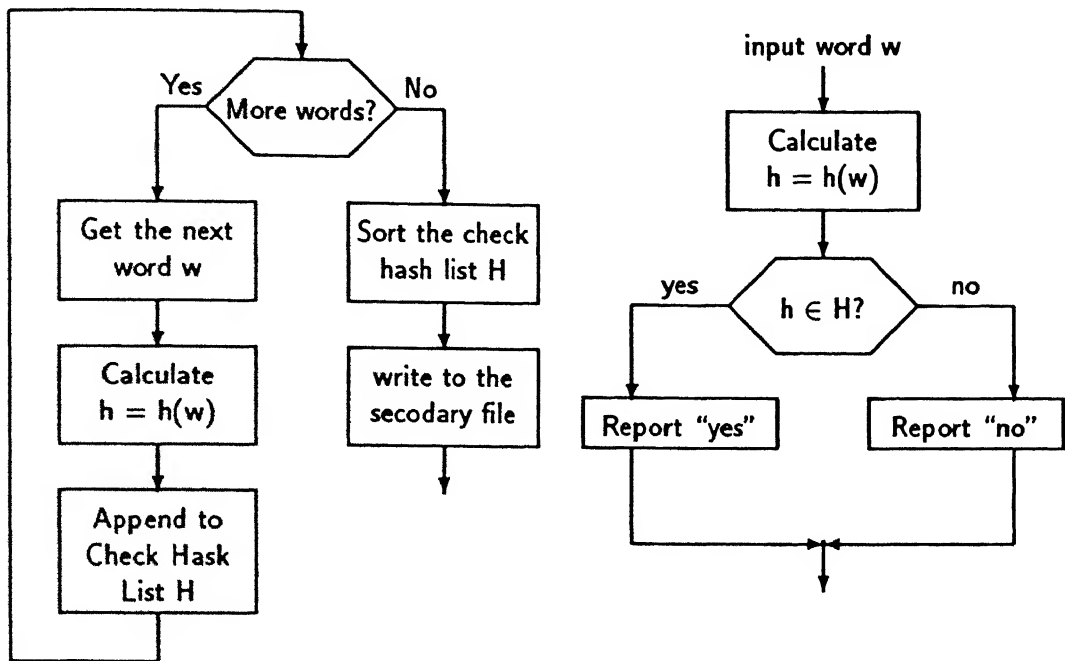


Figure 13: Lexicon – (a) Compilation: Building the Check Hash List (b) Analysis: Searching the Lexicon

Before any searching operation, the analyzer component reads this file into main memory. Now given a word x , its check hash $h(x)$ is calculated and the hash list is searched for $h(x)$ using binary search.

One should recognize the importance of the hash function in this approach. It should be sufficiently “random” and “uniform”. References [19, 26] provide a various techniques for designing good hash functions and present some hash functions. The hash function used in this implementation is given in Appendix C.

The following two tests are designed to evaluate the performance of the hash funtion.

1. One simple test is to count collisions during the lexicon creation. Surprisingly, when this hashing function is used for a preliminary Telugu lexicon having around 15000 words, it doesn't result in even a single collision.
2. Generally, a misspelled word would be a minor variation of the correct word and possible stems generated during stemming a given word would all differ in a character or two. So, a small program is written to produce minor variations of words in the lexicon by performing transformations like
 - changing a letter
 - inserting a new letter
 - deleting a letter
 - swapping two adjacent letters

This program was run on 1000 words randomly chosen from the words in the lexicon and generated a total of 8,32,196 words. The lexicon lookup of these words reported that 1560 of them were in the lexicon; 1421 really were, while 139 were not. Thus the algorithm showed up 0.017% error rate – quite reasonable in practice.

4.3 Others

The implementation of other parts of our content analysis scheme is relatively simple. This section provides short notes on these.

Exclusion Words Exclusion words are simply listed out in the exclusion words dictionary file. Each line may contain many words. There can be blank lines and comment lines (lines that start with a %) which are neglected. Words are collected into a simple sequential list and sorted. A binary search is performed on this list to identify if a given word is in EWD.

Synonyms Synonym Dictionary groups stems into concept classes. Each line in the file contains a list of equivalent stems. The first stem in the list is used as the concept identifier. As with many other linguistic files that we are using, comment lines or blank lines may appear anywhere in the SD file.

The file is read and compiled into a list of stem-concept pairs and sorted in the alphabetically increasing order of words. A binary search is performed to get the concept name associated with a given stem, if one such exists.

Phrases Phrase dictionary groups concept terms into phrases and gives them some mnemonic identifiers. Since different phrases may be synonyms, each mnemonic concept identifier may identify many phrases. Each phrase is specified as a sequence of at most 4 concept terms separated by spaces and enclosed in braces, like { reYlu pramAxaM } and each line contains a mnemonic identifier followed by a list of phrases. This file is read and compiled into a list having entries of the following structure:

```
/* Structure definition for entries in the list of phrases */
typedef struct phrase
{
    char    *concept;           /* the mnemonic identifier */
    char    *term[TERMS_PER_PHRASE]; /* the list of terms comprising
                                   the phrase */
} phrase_t;
```

This structure stores the information pertaining to a single given phrase.

In our phrase processing procedure, all phrase components must be present within the boundaries of the same sentence in a document before the corresponding phrase identifier is assigned to the document. No restriction is imposed on the order or position of the various

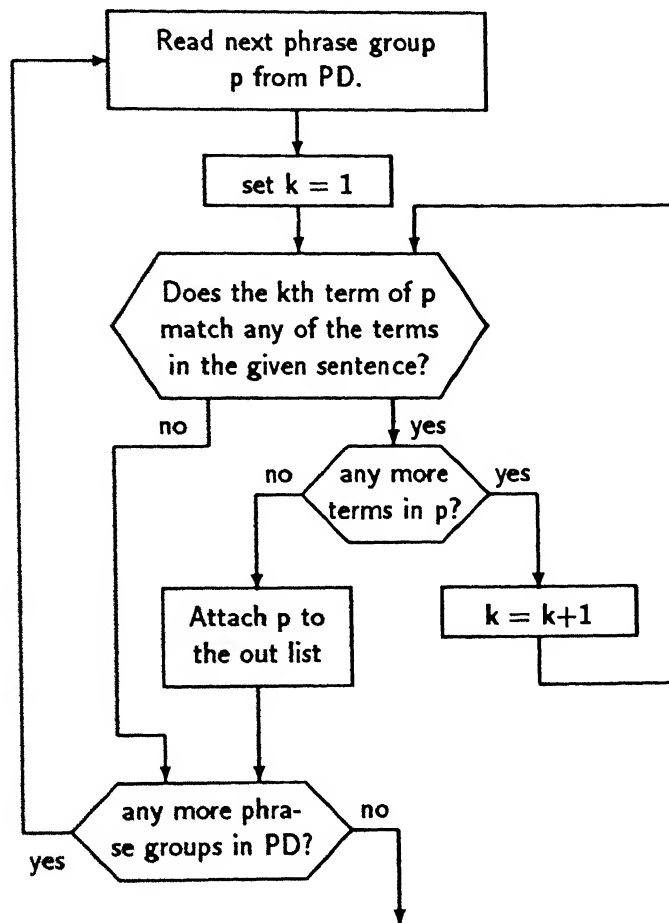


Figure 14: Phrase Detection and Matching

phrase components within a sentence. Accordingly, the algorithm for phrase finding is completely straightforward. It takes a sentence (in the form it appears after the exclusion word deletion, stemming and synonym translation) as input, and outputs the identifiers of the phrases recognized in the sentence. It proceeds using the general strategy given in Figure 14.

It considers the phrases one by one – matches the first component of a phrase with each term in the given sentence; the second component is then matched, and so on. For each phrase, the search terminates when either no further terms are present in the sentence, or when all components of the phrase have been processed, or when no matches are found in the given sentence for a given component. If a particular phrase is found in a sentence, the mnemonic identifier of the phrase is attached to the list of phrases recognized.

WAIS to Support Document Retrieval in Telugu

The Wide Area Information Servers (WAIS)[9] system is a set of products supplied by different vendors to help end-users find and retrieve information over networks. The information can be text, images, voice, or formatted documents. Thinking Machines, Apple Computer, and Dow Jones were the ones initially implemented this system for use by business executives. These products are becoming more widely available from various companies.

WAIS, at present, supports text retrieval only in English. However, it can be easily extended for other languages also. Our interest, in this thesis, is to make WAIS usable for text retrieval in Indian languages. It is easy to see that supplying a content analysis algorithm and designing a scheme for representing the foreign language script in Roman letters, any existing information retrieval system can be extended to support text retrieval in other languages.

To demonstrate the feasibility of this technique and to make available a practical document retrieval system for Indian languages, we attempted to extend WAIS to support document retrieval in Telugu. In this chapter, we first discuss the components and features of WAIS that are of interest to us, and then we describe the major difficulties faced and solutions used in the integration process.

5.1 An Overview of WAIS

The WAIS system architecture has three major components:

- the user workstation
- the servers
- the communication protocol

Servers are accessible on a network. A server has some database of documents that can be queried and retrieved from. Servers may also have an additional database containing information about other servers. Servers accept the queries from the user workstations and try to find documents that contain those words and phrases and rank them based on heuristics. Workstations find appropriate information for the user by contacting, probing, and negotiating with information servers. They communicate with the servers using an extended version of a standard protocol Z39.50[48].

We are interested only in the way the servers identify the relevant documents for given queries. Other parts of the architecture like the protocol and the clients or user interfaces are not of an immediate interest. The following paragraphs describe briefly the indexing and searching schemes of WAIS.

Index Record

WAIS uses an inverted indexing scheme that is not very different from any existing indexers. The *index record* for each document collection comprises of six secondary files. For a document collection named "sample" these files would be -

sample.inv	- the index file
sample.dct	- the dictionary file
sample.doc	- the document table file
sample.fn	- the file name table file
sample.hl	- the headlines file
sample.src	- the source description file

The Source Description File The source description file stores information on the document collection. It is used for accessing the document collection. It contains information like how to contact the server of this document collection, the name to use to refer to the document collection, the service cost, and a list of important keywords of the document collection to give an idea of the subject area of the document collection. This is an ASCII file and additional information may be put into it if needed.

The File Name Table File The document store consists of a set of secondary files. The file name table file contains information of the different files in the document store. The information includes a file name, its type (like a text file or an image file etc), the write date of the file etc.

The Document Table File In WAIS, each file in the document store may contain a single document or multiple documents. The document table file, for a particular document collection, provides information on all the documents in the collection. Each entry corresponds to one document and contains information like the name of the file that contains this document, position in the file where this document starts, the length of the document in number of characters etc.

The Headlines File WAIS attaches to each document a headline; this headline can, for example, be the title of the article if the document is a newspaper or journal article. This file stores the list of headlines one corresponding to each document. As a response to a user query, the headlines of the relevant documents are given. Then the user may select the documents that he finds interesting which are retrieved from the document store and presented to the user.

The Index File This file is the core of the index record. It contains a set of "postings" one corresponding to each index term. Each posting is an index term followed by a list of entries that represent in which documents this term has occurred and the corresponding weight.

The Dictionary File The index term entries in the index file are of varying length. This presents difficulties during searching. To speed up the searching process an additional dictionary file is used. This file uses a 2-level B-tree data structure. Each entry contains an index term and a pointer to the entry in the index file that corresponds to this index term. Each entry is, thus, of fixed length. The entries are in alphabetical order of terms.

Indexing

The indexer is composed of front-end and back-end. The front-end parses the input file for words and creates the filename, headline and document table entries during parsing. The words are then passed to the back-end.

The back-end accumulates words in a memory hashtable. This hashtable is flushed after accumulating certain number of words to an intermediate index file. All these intermediate files are merged at the end into the existing index file of the document collection and the dictionary file is modified accordingly.

Searching

The search engine also has a front-end and a back-end. A given user query is parsed for its keywords by the front-end. It then passes these keywords to the backend. The backend searches the dictionary file for these keywords and the "postings" corresponding to each of the keywords are then obtained from the index file. These postings give the information on in which documents the keywords occur and the corresponding weights. The front-end then uses this information to rank the documents. Following a lookup of headline, document table and file name files, the details of the high ranking documents are handed over to the user.

5.2 Integration

In this section, we look into the issues and problems that arose in extending WAIS to support document retrieval in Telugu.

We aim at making as few changes as possible to the existing WAIS software. Also, whatever changes we make should not affect the performance of the system at other places.

Encoding. For the purpose of indexing and searching, WAIS converts the words in the input documents and queries into small case letters. The hashtable scheme used by the back-end of the indexer is based on assumption that the words contain only small case letters and the index and dictionary file structures are also designed to store words in small case letters only. But with our Roman encoding scheme for Indian languages, both capital letters and small case letters are used to represent letters of Indian languages. Hence, to be able to use WAIS for Indian languages also, we have to modify the code at all these places to allow capital case letters also. But the aim is to avoid such major changes to the code.

This problem is solved by an additional encoding/decoding step to the front-ends of the indexer and the search engine. In this step, the words in Indian language are converted to a representation which uses small case Roman letters only. This conversion step is simple, the letters which are already in lower case are copied as they are and the letters in upper case are replaced by the corresponding lower case letter followed by a *z*. This conversion process is reversible. For example, the word *AyurvExaM* will be converted to *azyurvezxamz* and *hOmioPawi* to *hozmiyozpawi*.

Content Analysis. WAIS itself does not use any complicated content analysis procedures for English documents. Few changes to the front-ends of the indexer and search engine were required to integrate the content analysis procedures for Indian languages into WAIS. When indexing (or searching for) documents in Telugu, the front-end passes the documents (or queries) to the content analysis procedure shown in Figure 3 which returns a set of keywords. These keywords are then passed as usual to the back-end after converting them to the lower case letter representation.

Marking Telugu Words. A document collection or database may contain both English and Telugu documents. In such cases, it becomes necessary to distinguish between the English and Telugu words stored in the index file and the dictionary file. Otherwise, the system may retrieve an English document as a response to a Telugu query. To make it possible to differentiate between words from different languages, an additional field into the dictionary and index file structures has to be introduced. This requires changes at many places in the WAIS code. This is not attempted primarily because of two reasons:

- It is generally not the case that a same document collection will contain multiple language documents.

- It is too rare that a proper Telugu (or any Indian language) word represented in the lower case letter scheme has a letter sequence that constitutes a proper English word.

Changes to Query Format. WAIS uses the standard Z39.50 protocol with few extensions for the communication between clients and servers on the network. The packet format that a client uses to send a search request to a server has fields for the document collection name, the query, etc., but it does not have any field that can be used to specify the language of the query. But the server has to know whether a particular search request is in English or Telugu; it applies content analysis procedure if the search is in Telugu. Alternative is to perform the content analysis and the encoding to lower case representation at the client itself and send the resulting keywords as the query in the packet. This solution is not feasible because it requires the client also have the content analysis software and the different linguistic files. Since each document collection may have a different set of linguistic data files, such solution is impractical.

The solution adopted is to append the string "<telugu>" at the front for the queries in Telugu before they are written into packets. On the server side, the front end of the search engine knows that the query is in Telugu by seeing the string "<telugu>" at the beginning of the query. It then can apply the content analysis and transliteration to lower case representation before the keywords are searched in the index and dictionary files.

No other major modifications to the WAIS code are required. The result is a practical and immediately usable distributed information retrieval system for Telugu. To use this system for any other Indian language, the only thing one has to do is to supply the required linguistic data files and dictionaries for that language.

Conclusions

In this concluding chapter, we briefly summarize the work reported in this thesis; we also consider various extensions possible to this work and identify places for further reasearch and exploration.

6.1 Summary of This Work

In this section, we provide a brief summary of the contributions of this work. In this thesis, we aimed at making available a practical information retrieval system for Indian languages. By designing a content analysis scheme suitable for texts in Indian languages and scheme for representing Indian languages' text in Roman letters, any existing information retrieval system can be easily extended to support retrieval in Indian languages also. We demonstrated this by extending WAIS to support text retrieval in Telugu.

For the content analysis of documents in Indian languages, we designed a four step process comprising of -

- Common word deletion
- Stemming
- Synonym recognition
- Phrase detection.

Simple and straight-forward approaches, that are proven to perform reasonably well, are adapted from literature and earlier implemented systems, for the latter three steps. In the case of stemming, none of the existing techniques could be directly used for Indian languages, because of their relatively complex morphological structure. In this work, we developed and implemented a space-efficient and fast stemming algorithm for Indian languages.

The implemented content analysis procedure is then integrated into WAIS to make it support document retrieval in Indian languages. The final system is then thoroughly

tested for Telugu. The system can be directly used for any other Indian language by providing the required linguistic data files, namely a list of common words, a list of word suffixes and recoding rules (or paradigm tables), the lexicon, a synonym dictionary and a phrase dictionary. No changes to the code are required.

6.2 Scope for Further Work

This work is just a first step towards a new direction and can be continued in several ways. Many immediate and straight-forward extensions are possible and there is a lot of scope for further study.

Statistical Data Generation. In preparing a common word dictionary, a synonym dictionary, and a phrase dictionary, we suggested that the system manager may use various statistical information on the particular document collection of interest. Various statistically based measures for 'term significance', 'term synonymity' and 'term co-occurrence' have been suggested [42, 43, 45, 46]. A survey of these statistical measures could be performed and programs could be developed for generating various statistical data of interest to the system manager.

Syntax-based Phrase Recognition. The phrase recognition scheme used in this implementation is not fully automatic, in the sense that, it is based on a dictionary of candidate phrases developed a priori by the system manager. Alternatively, completely automatic syntax based phrase recognition procedure could be used. Literature suggests various possible approaches in this direction[13, 22]. The applicability of these syntax based techniques for generating phrases, instead of single words, as index terms, for Indian languages has to be studied. In particular, approaches that do not place additional burdens on the lexicon (for example, lexicon supplying syntactic information on root words) have to be studied.

We suggest here one possibility that can be explored further. The motivation for this approach comes from combining the ideas of [24], [16, 17] and [8]. The technique involves using a dictionary of permissible syntactic formats for phrases. For example, we can specify that a sequence of 2 or more nouns or a sequence of one or more adjectives followed by a noun as some permissible formats. The phrase recognition procedure involves identifying a sequence of words in the document or query text that confirms to a permissible syntactic format as specified in this dictionary. The required syntactic information on words can be obtained without using a lexicon that supplies this information, using a two step algorithm as follows[8, 16, 17]:

In the first step, the words of the original text are treated in isolation, in the second step the context is taken into consideration. The first step uses the fact that some word suffixes can be used to characterize the part-of-speech of the words, e.g., in English, words ending with `able` are mostly adjectives like `probable`, words ending with `fy` are mostly verbs like `modify` and words ending with `ure` are mostly nouns like `closure`. There are, ofcourse, exceptions to these regularities (e.g., `disable` is a verb) which can be handled

as exceptions. The suffix dictionary can contain such syntactic information along with the suffixes.

Though in the first step, a certain recognition of parts-of-speech is possible, the second step can obtain more useful information. This step handles the words whose syntactic information could not be obtained in the first step, based on a set of context rules. For example, in English, one can give rules like this:

$$\begin{array}{ll} \text{art } Z \text{ } F & \Rightarrow \text{art } N \text{ } F \\ A \text{ } Z \bullet & \Rightarrow A \text{ } N \bullet \end{array}$$

where Z means "unknown part-of-speech", F is a function word, art an article, A an adjective, N a noun. Thus, the first rule would recognize a word w in the context, e.g., "the w is .." as a noun, and the second rule would recognize w in "... good w ." as a noun.

The applicability of such techniques and the required rules have to be further studied. Phrase recognition by such approach has an advantage that, it does not require any prior set up phrase dictionaries of the kind we used, which implies that it can be used irrespective of the subject field of the document collection.

Error Evaluation of the Stemming Algorithm. The stemming algorithm implemented shows up a reasonably good performance in terms of the memory requirements and speed. It is fast enough because all the required data (i.e., the suffix trie, recoding rules etc.) lies in the main memory except for the lexicon, if it is too large for a particular language; and the main memory and disk space requirements are also not much. However, a study of the rate of erroneous performance of this algorithm and the corresponding effects on the system performance could be taken up. Such a study may help to compare the efficacy of different possible stemming algorithms or to indicate the most fruitful places for further improvements.

There are two kinds of stemming errors to consider: *overstemming* and *understemming*[32]. The former occurs when two unrelated words are stemmed to a common root and results in the retrieval of irrelevant documents through matching of unrelated terms. The second kind of error occurs when a word is not stemmed to a root to which it should be, and leads to omission of relevant material by not matching related terms. While the former results in lower precision, the latter leads to lower recall.

The performance of the algorithm could be studied in stages – how it performs if no recoding step is used, if no lexicon lookup is used etc. – to observe the importance of various steps involved in the algorithm. This study could be performed on various languages and steps that really do not improve the performance could be deleted for a particular language.

Performance Evaluation. The system performance could be studied conducting large scale experiments. Such a study should start with a large collection (at least 500) of documents on a particular subject area and all the required linguistic data files. Parameters like precision and recall that quantify the retrieval effectiveness are of particular interest. To base experiments on these parameters, one should have a priori a list of queries and a list of documents in the document collection that are relevant to those queries. Various parameters can be evaluated on each of the queries and can be averaged. Such an experiment can serve as a feasibility study and can also provide a base for identifying the possibilities for various improvements.

No A.118778

Bibliography

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Reading, Massachusetts: Addison-Wesley, 1986.
- [2] R. Attar, Y. Choueka, N. Dershowitz and A. S. Fraenkel, *KEDMA - Linguistic Tools for Retrieval Systems*, Journal of the ACM, Vol. 25, No. 1, January 1978, pp. 52-66.
- [3] Allen I. Holub, *Compiler Design in C*, Prentice-Hall of India, 1993.
- [4] Allen Kent, *Information Analysis and Retrieval*, Becker And Hayes Inc., 1971.
- [5] Bentley, J., *A Spelling Checker*, Communications of the ACM, Vol. 28, No. 5, May 1985, pp. 456-462.
- [6] Bharati Akshar, Vineet Chaitanya, and Rajeev Sangal, *Natural Language Processing: A Paninian Perspective*, Prentice-Hall of India, 1994.
- [7] Boune C. P., *Frequency and impact of spelling errors in bibliographic databases.*, Information Processing and Management, Vol. 13, No. 1, Jan 1977, pp. 1-12.
- [8] Braun S., *Linguistically Based Methods for Indexing*, Proceedings of the IFIP-INFOPOL International Conference on Information Processing, Poland, March 1976, North-Holland Publishing Company, 1977. pp. 187-203.
- [9] Brewster Kahle, *Wide Area Information Servers Concepts*, Documentation supplied with WAIS software, Thinking Machines, 1989.
- [10] Carolyn J. Crouch and Donald B. Crouch, *An Analysis of Document Retrieval Systems Using a Generalized Model*, Proceedings of the Fourth International Symposium on Computer and Information Sciences, December 1972.
- [11] Cleverdon, C. W., Mills J., and E. M. Keen, *Factors determining the performance of Indexing Systems*, Vol. 1 - Design, Aslib Cranfield Research Project, Cranfield, 1966.
- [12] Comparative Systems Laboratory, *An Inquiry into Testing of Information Retrieval Systems: Final Report*, Centre for Documentation and Communication Research, Case Western Reserve University, 1968.
- [13] Christine A. Montgomery, *Linguistics and Information Science*, Journal of the ASIS, May-June 1972, pp. 195-219.

- [14] David C. Blair and M. E. Maron, *An Evaluation of Retrieval Effectiveness for a Full-text Document-Retrieval System*, Communications of the ACM, Vol. 28, No. 3, March 1985, pp. 289-299.
- [15] Dodds, D. J., *Reducing Dictionary Size by Using a Hashing Technique*, Communications of the ACM, Vol. 25, No. 6, June 1982, pp. 368-370.
- [16] Earl, Lois L., *Part-of-Speech Implications of Affixes*, Mechanical Translation and Computational Linguistics, Vol. 9, No. 2, June 1966.
- [17] Earl, Lois L., *Automatic Determination of Parts of Speech of English*, Mechanical Translation and Computational Linguistics, pp. 53-67.
- [18] Gonnet, G. H., and Baeza-Yates, R., *Handbook of Algorithms and Data Structures, In Pascal and C*, Second Edition, Addison-Wesley, 1991.
- [19] Josef Pieprzyk and Babak Sadeghiyan, *Design of Hashing Algorithms*, LNCS-756, Springer-Verlag, 1993.
- [20] Julia S. Falk, *Linguistics and Language: A survey of basic concepts and Implications*, John Wiley & Sons, 1973.
- [21] Karen Sparck Jones and E. O. Barber, *What Makes An Automatic Keyword Classification Effective?*, Journal of the American Society for Information Science, May-June 1971, pp. 166-175.
- [22] Karen Sparck Jones and Martin Kay, *Linguistics and Information Science*, Academic Press, 1973.
- [23] Kingbiel, P. H., *A Technique for Machine-Aided Indexing*, Information Storage and Retrieval, Vol. 9, No. 2, September 1973, pp. 477-494.
- [24] Klingbiel, P. H., *Machine-aided Indexing of Technical Literature*, Information Storage and Retrieval, Vol. 9, pp. 79-84, 1973.
- [25] Klingbiel, P. H., and Rinker, C. C., *Evaluation of Machine-Aided Indexing*, Information Processing & Management, Vol. 12, pp. 351-366, 1976.
- [26] Knuth, D.E., *The Art of Computer Programming, Vol. III - Sorting and Searching*, Addison-Wesley, 1973.
- [27] Krishnamurthi B., *Telugu Verbal Bases: A Comparative and Descriptive Study*, Motilal Banarsidass, 1972.
- [28] Krishnamurthi and Gwynn, *A grammar of modern Telugu*, Oxford University Press, 1985.
- [29] Lancaster, F. W., *On the Need for Role Indicators in Post-Coordinate Retrieval Systems*, American Documentation, Vol. 19, pp. 42-46, 1968.

- [30] Lancaster, F. W., *Criteria by which Information Retrieval Systems may be Evaluated*, in *Information Retrieval Systems – Characteristics, Testing and Evaluation*, 2nd Edition, Jon Wiley and Sons, 1979.
- [31] Lovins, J. B., *Development of a Stemming Algorithm*, *Machanical Translation and Computational Linguistics*, Vol 11, 1968, pp. 22-31.
- [32] Lovins, J. B., *Error Evaluation for Stemming Algorithms as Clustering Algorithms*, *Journal of the ASIS*, January-February 1971, pp. 28-40.
- [33] McIlroy, *Development of a spelling list*, *IEEE Transactions on Communications*, COM-30, No. 1, January 1982, pp. 81-92.
- [34] Meadow, C., *Applied Data Management*, John Wiley and Sons, Inc., New York, 1976.
- [35] Nick Cercone, *Morphological Analysis and Lexicon Design for Natural Language Processing*, *Computers and the Humanities*, Vol. 11, pp. 255-258, 1978.
- [36] Pacak, M. G., Pratt, A. W., and White, W. C., *Automatic Morphosyntactic Analysis of Medical Language*, *Information Processing and Management*, Vol. 12, pp. 71-76, 1976.
- [37] Peterson, J. L., *Computer Programs for Detecting and Correcting Spelling Errors*, *Communications of the ACM*, Vol. 23, No. 12, December 1980, pp. 676-687.
- [38] Peter Robinson and Dave Singer, *Another Spelling Correction Program*, *Communications of the ACM*, Vol. 24, No. 5, May 1981, pp. 296-297.
- [39] Ramagopala Krishnamurthi, K., *Telugu Vyakaranam*, Sri Sailaja Publications, 1993.
- [40] Richard Sproat, *Morphology and Computation*, The MIT Press, 1992.
- [41] Robert Nix, *Experience With a Space Efficient Way to Store a Dictionary*, *Communications of the ACM*, Vol. 24, No. 5, May 1981, pp. 297-298.
- [42] Salton, G. and Lesk M. E., *The SMART Automatic Documet Retrieval System*, *Communications of the ACM*, Vol. 8, No. 6, June 1965.
- [43] Salton, G., *Automatic Information Organization and Retrieval*, McGraw-Hill Book Company, 1968.
- [44] Salton, G., ed., *The SMART Retrieval System - Experiments in Automatic Document Processing*, Prentice-Hall Inc., 1971.
- [45] Salton, G., *Dynamic Information and Library Processing*, Prentice-Hall Inc, 1975.
- [46] Salton G., and Michael J. McGill, *Introduction to Moderen Information Retrieval*, McGraw-Hill International Book Company, 1983.
- [47] Wiederhold, G., *Database Design*, McGraw-Hill Book Company, New York, 1977.
- [48] *ANSI Z39.50 Version 2*, Documentation supplied with WAIS, Thinking Machines, 1991.

APPENDIX A

The Encoding Scheme

The modern Telugu alphabet consists of 51 letters, 16 of which are vowels and the others are consonants. Using both upper and lower case Roman alphabet, we have 52 letters. Hence a one-to-one mapping of Telugu letters to English letters is possible. But, it is better to have a representation scheme that represents Telugu letters in the corresponding orthographically related letters in English. Such a scheme is easy to use and fast to learn. The encoding scheme given here is based on the scheme used in *Anusaraka*[6].

a	A	i	I	u	U	q	Q	
అ	ఆ	ఇ	ఈ	ఉ	ఊ	ఋ	ౠ	
e	E	eY	o	O	oY	M	H	
ఎ	ఏ	ఐ	ఒ	ఓ	ఔ	అం	ఆః	
k	K	g	G	f	c	C	j	J F
క	ఖ	గ	ఘ	ఙ	చ	ఛ	జ	ఝ ఞ
t	T	d	D	N	w	W	x	X n
ట	ఠ	డ	ఢ	ణ	త	థ	ద	ధ న
p	P	b	B	m				
ప	ఫ	బ	భ	మ				
y	r	v	l	L	s	S	R	h kR
య	ర	వ	ల	ళ	స	శ	ష హ	క్ష

Figure 15: The Encoding Scheme for Telugu

APPENDIX B

Sample Dictionaries

This appendix gives a sample set of various linguistic dictionaries, identified to be of use in the context of document retrieval systems in this work. The main purpose of this is to give a more clear picture of the specific formats used for various dictionary files. It also provides an approximate estimation of the size of such dictionaries required for a particular language.

No samples are given for dictionaries like exclusion words dictionary, synonym dictionary, lexicon etc., which use simple formats, as explained in Chapter 4.

Suffix Dictionary

Compiling an almost complete set of word suffixes for Telugu is one major part of this work. A preliminary list of simple word suffixes (of the kind that is required for an iterative suffix truncation algorithm) is based on the Telugu grammar presented in [27, 28]. This list is then compiled into a one-class list of suffixes as required by our longest-match algorithm. The resulting set of suffixes is then tested against a large collection of words from various magazine and newspaper articles and books. The present experimental list contains about 750 suffixes, mostly that get attached to verb and noun roots. This resulted in an inverse suffix trie of about 1750 nodes. Each node being represented by a vector of 53 two-byte elements, the trie occupies about 185 Kbytes of memory space.

Shown below is a partial set of verb suffixes from the developed suffix dictionary for Telugu.

```
% .....  
%                               Verb Suffixes  
% .....  
% .....
```

```
% Almost every Telugu verb has a finite form and a non-finite form. A  
% finite form is one that can stand as the main verb of a sentence and  
% occur before a final pause (full stop). A non-finite form cannot stand  
% as a main verb and rarely occurs before a final pause. The verb suffix-  
% es are, accordingly, classified as ones that attach to finite-forms and  
% ones that attach to non-finite-forms.
```

```
% -----  
%                               Suffixes that attach to finite form verbs  
%  
%
```

% Most of the finite form verbs can be analyzed into :

% verb stem + tense suffix + personal suffix.

% Tense suffixes depend on the tense. And personal suffixes get attached based on the person, number and gender of the subject of the sentence.

% -----

% Past tense suffixes with personal suffixes attached to them

050 Anu Enu Avu Evu Adu Edu iMxi iMxE inaxi inaxE
Aru Eru Amu AM Emu EM Ayi

051 nAnu nAvu nAdu niMxi niMxE ninaxi inaxE
nAmu nAM nAru nAyi

052 dAnu dAvu dAdu daxi daxE
dAmu dAM dAru dAyi

% Future-habitual tense suffixes with personal suffixes attached to them.

% (The future-habitual tense is so called because it has two meanings.

% It can express an action or state that will take place in the future,

% or an action or state that is habitual.

% Eg. Avulu pAlu iswAyi.

% can mean either 'The cows will give milk' or 'Cows give milk'.)

060 wAnu wAvu wAdu wuMxi wOMxi wOMxO
wAmu wAM wAru wAyi

061 tAnu tAvu tAdu tuMxi tOMxi tOMxO
tAmu tAM tAru tAyi

% Negative suffixes with personal suffixes attached to them.

% (The formation of a verb paradigm in the negative tense rather than the

% use of a separate word or particle negation as in English, Hindi and

% many other languages. Negative verbs of this type are in the future

% habitual tense and negate the affirmative verb occurring in this tense,

% corresponding to English 'does not.. / will not..'

070 anu avu adu axu
amu aM aru
anuMxi anunnAru anunnAvu anunnAmu anunnAM

071 nu vu du xu
mu M ru

% Imperative suffixes - The imperative forms of verbs carry two suffixes:

% 2nd person singular and 2nd person plural and they occur in both the

% affirmative and the negative. The meaning conveyed by the imperative

% in the singular is informal or impolite. The imperative plural is used

% for politeness when addressing one person or without any reference to

% degrees of politeness when addressing a number of person.

070 u aMdi
aku akaMdi

071 Mdi
ku kaMdi

% Hortative suffixes - Hortative is an imperative that includes the
% speaker or the addresser also. In English, it occurs in expressions
% like 'let us go', 'let us do' etc. This kind of expression is conveyed
% in Telugu by adding to the verb stem the hortative suffix 'xA' followed
% by the first person plural suffix 'mu' or 'M'

062 xAM xAmu

% Durative suffixes - Verbs in durative form express the aspect of
% 'continuity' of an action or state in the present, past or future.
% A durative finite verb has the following constituents -
% basic stem + w/t (durative suffix) + un ('to be')
% + tA/tuM (tense suffix) + personal suffix

060 wuMtAnu wuMtAvu wuMtAdu wuMtuxi
wuMtAru wuMtAmu wuMtAM wuMtayi

061 tuMtAnu tuMtAvu tuMtAdu tuMtuxi
tuMtAru tuMtAmu tuMtAM tuMtayi

% Polite Imperative suffixes - Finite verbs in polite imperative convey
% a polite request; they are used only in the case of an addressee with
% whom the speaker is familiar. A polite imperative finite verb has
% the following constituents -
% hortative stem + the tense-mode suffix 'xu'
% + 2nd person suffixes 'vu' and 'ru'

062 xuvu xuru

% -----
% Suffixes that attach to non-finite verbs
%
% Non-finite verbs can be classified into two types of participles -
% Conjunctive participles - end subordinate verb clauses
% Relative participles or verbal adjectives - end subordinate
% adjectival clauses.
%
% There are four conjunctive participles -
% Past/Perfective
% Durative
% Conditional
% Concessive
% Each of these have both affirmative and negative forms.
% -----

% Past/Perfective participle refers to the completion of an action which

% precedes in point of time the action denoted by the finite verb. The
 % past participle is formed as follows -
 % verb stem (which occurs with past tense suffixes) + 'i'

070 aka akuMda

071 ka kuMda

% Durative Participle is used when the action in the subordinate clause
 % is simultaneous with that in the main clause. Formed as follows -
 % verb stem (in the form that occurs in durative finite form)
 % + wU / tU

080 wU wUnE

081 tU tUnE

% Conditional Participle ends conditional clauses. Formed as follows:
 % verb stem (in the form that occurs in the future habitual)
 % + wE / tE
 % The meaning of this form corresponds to 'if' or 'when' in English.

060 wE wEkAni wEgAni wEnEgAni

061 tE tEkAni tEgAni tEnEgAni

% Concessive Participle corresponds to 'although, even though, even when,
 % even if' in English. It has a present/past meaning; for example,
 % 'Ayana vaccina' may mean either 'although he comes' or 'although
 % he came'. Forms as follows:
 % stem (in the past tense form) + inA / nA

060 i ina inA inatlu

061 na nA natlu

070 ani anatlu

% -----
 % The Infinitive Form of Verbs
 % The infinitive is not as common in Telugu as it is in English. It
 % generally occurs -
 % 1. before the nouns 'akkara', 'avasaraM' and 'pani'
 % 2. before the suffixes 'E' (emphatic), 'gA' and 'batti'
 % 3. at the end of a sentence to form a special type of finite verb
 % 4. in compound verbs.
 % The infinitive forms as follows -
 % verb stem (which occurs in the negative tense) + 'an' / 'n'
 % -----

% Infinitive suffix followed by nouns 'akkara', 'avasaraM' and 'pani'

070 anakara anavasaraM anavasaramu abani

anakaraExu anvasaraMExu anavasaramulExu abanilExu abalExu

071 nakkara navasaraM navasaramu nabani

B. Sample Dictionaries

nakkaraLExu navasaraMlExu navasaramulExu nabanilExu nabaLlExu

% Infinitive suffix followed by the suffix 'E'. This form is used to
% express emphatic negation in the verb, as in the following sentences:

% vAdu akkada 'uMdanE' uMtAdu.

% Ame 'rAnE' vacciMxi.

% This type of sentences are peculiar to Dravidian languages.

070 anE

071 nE

% Infinitive suffix followed by the suffixes 'gA'/'gAnE'; such a sequence
% gives the meaning 'when as', 'as soon as' etc.

070 agA agAnE

071 gA gAnE

% Suffixes to take care of the most frequent compound verbs. Essetially
% this list is not required if compound analysis is done as part of the
% stemming.

060 wunna wunnA wunnatlu wunnAyi wunnAru wunnAdu wunnAvu wunnAnu wunnaxi

061 tunna tunnA tunnatlu tunnAyi tunnAru tunnAdu tunnAvu tunnAnu tunnaxi

070 alExu

Aka

Ali AlA Alsi AlsiMxi AlsiMxA AlsocciMxi AlsocciMxA AlsivacciMxi

AlsivacciMxA AlsiMxiga

avalasi avalasiMxi avalasiMxA avalasiMxiga avalasocciMxi avalasocciMxA

avalasivacciMxi avalasivacciMxiga

avaxxu oxxu

agalugu agalagadaM agalagataM

agalanu agalavu agaladu agalaxu

agalamu agalaM agalaru

agalagAli

alEnu alEnA alEvu alEvA alEdu alEdA alExu alExA

alEmu alEM alEmA alEru alErA alEka alEkunna alEkuMtE

avaccu avaccunu avaccuna avacca occu occunu occuna occA

akapOvu akapOnu akapOdu akapOxu akapOmu akapOru

akapOyAvA akapOyAnA akapOyAdA akapOxA akapOma akapOrA

akapOvaccu akapOvE akapOyinA

alEkapOvu alEkapOvadaM alEkapOvataM

alEkapOyAnu alEkapOyAvu alEkapOyAdu alEkapOyiMxi

alEkapOyAnA alEkapOyAvA alEkapOyAdA alEkapOyiMxA

alEkapOyAmu alEkapOyAM alEkapOyAru alEkapOyAyi

alEkapOyAmA alEkapOyArA alEkapOyAya

alEkapOvE alEkapOyinA

akUdaxu akUdaxA agUdaxu agUdaxA

abOvu abOvadaM abOvataM

abOwunnaNu abOwunnaVu abOwunnaDu abOwuMxi

abOwunnaMu abOwunnaM abOwunnaRu abOwunnaYi

abOyAnu abOyAvu abOyAdu abOyiMxi
 abOyAmu abOyAm abOyAru abOyAyi
 anivvu anivvadaM anivvataM
 aniswAnu aniswAvu aniswAdu aniswuMxi
 aniswAmu aniswAM aniswAru aniswAyi
 aniccAnu aniccAvu aniccAdu anicciMxi
 aniccAmu aniccAM aniccAdu aniccAyi
 anivvu anivvaku anivvaMdi anivvakaMdi
 abadu abadataM abadadaM
 abaddAnu abaddAvu abaddAdu abdiMxi
 abaddAmu abaddAM abaddAru abaddAyi
 abadawAnu abadawAvu abadawAdu abadiMxi
 abadawAmu abadawAM abadawAru abadawAyi
 abadu abadaku abadaMdi abadakaMdi
 abadunu

071 lExu

vAli vAlA vAlsi vAlsiMxi vAlsiMxA vAlsocciMxi vAlsocciMxA
 vAlsiivcciMxi vAlsiivacciMxA
 valasi valasiMxi valasiMxA valasiMxiga valasocciMxi valasocciMxA
 valasivacciMxi valasivacciMxiga
 yAka
 vaxxu
 galugu galagataM galagadaM galagAli
 galanu galavu galadu galaxu
 galamu galaM galaru
 lEnu lEnA lEvu lEvA lEdu lEdA lExu lExA
 lEmu lEM lEma lEru lErA lEka lEkunna lEkuMte
 vaccu vaccunu vaccuna vacca
 kapOvu kapOnu kapOdu kapOxu kapOmu kapOru
 kapOyAvA kapOyAnA kapOyAdA kapOxA kapOma kapOra
 kapOvaccu kapOwE kapOyina
 lEkapOvu lEkapOvataM lEkapOvadaM
 lEkapOyAnu lEkapOyAvu lEkapOyAdu lEkapOyiMxi
 lEkapOyAnA lEkapOyAvA lEkapOyAdA lEkapOyiMxA
 lEkapOyAmu lEkapOyAM lEkapOyAru lEkapOyAyi
 lEkapOyAmA lEkapOyArA lEkapOyAyA
 lEkapOwE lEkapOyina
 kUdaxu kUdaxA gUdaxu gUdaxA
 bOvu bOvadaM bOvataM
 bOwunnaAnu bOwunnaAvu bOwunnaAdu bOwuMxi
 bOwunnaAmu bOwunnaAM bOwunnaAru bOwunnaYi
 bOyAnu bOyAvu bOyAdu bOyiMxi
 bOyAmu bOyAm bOyAru bOyAyi
 nivvu nivvadaM nivvataM
 niswAnu niswAvu niswAdu niswuMxi
 niswAmu niswAM niswAru niswAyi
 niccAnu niccAvu niccAdu nicciMxi
 niccAmu niccAM niccAdu niccAyi
 nivvu nivvaku nivvaMdi nivvakaMdi
 badu badadaM badataM
 baddAnu baddAvu baddAdu badiMxi

baddAmu baddAM baddAru baddAyi
 badawAnu badawAvu badawAdu badiMxi
 badawAmu badawAM badawAru badawAyi
 badu badaku badaMdi badakaMdi
 badunu

050 E EMxuku

% -----
 % Suffixes that add to verb stems to result in verbal nouns
 %
 % The verbal nouns are formed by adding the suffix ataM/adaM to the form
 % of a verb stem which occurs in the negative tense. Since the resulting
 % words become nouns they can take almost all the noun suffixes.
 % The following list thus contains the verbal noun suffix 'ataM/adaM'
 % followed by all the applicable noun suffixes.
 % -----

070 ataM adaM
 ataMwO adaMwO
 atAniki adAniki
 ataMlO adaMlO
 atamE adamE

071 vataM taM vadaM daM
 vataMwO vadaMwO taMwO daMwO
 vatAniki vadAniki tAniki dAniki
 vataMlO vadaMlO taMlO daMlO
 vatamE vadamE tamE damE

Paradigm Tables

The developed paradigm tables dictionary comprises of about 75 noun and verb paradigm categories. This list when compiled, generated about 500 recoding rules. The verb paradigms from compiled paradigm tables dictionary is given below.

% -----	# win
% Verb Paradigms	win : 050, 051, 070
% -----	wiM : 061, 062
 # amu	 # cEsukon
amu : 060, 062	cEsukon : 051, 050
amu : 050, 070	cEsukO : 071
	cEsukoM : 061, 062
 # adug	 # icc
adugu : 060, 062	is : 060
adig : 050	ix : 062
adag : 070	icc : 050
adug : 070	ivv : 070
 # wagul	 iyy : 070

kAlc	:	050, 070	waM	:	060, 062
			wann	:	050, 070
# cUpiMc			# veLL		
cUpis	:	060	veL	:	060, 062
cUpix	:	062	veLL	:	050, 070
cUpiMc	:	050, 070			
# kott			# pad		
koda	:	060	pad	:	052, 070
kodu	:	060, 062	pada	:	060, 062
kott	:	050, 070	padu	:	060, 062
# cepp					
ceba	:	060			
cebu	:	060, 062			
cepp	:	050, 070			

Compiling the above set of paradigm tables file entries results in the following list of stem recoding rules.

cat	old	new	minroot	cat	old	new	minroot
code	end	end	size	code	end	end	size
50	S	c	1	62	M	n	1
50	S	y	1	62	M	nn	1
50	ic	uc	2	62	a	NULL	1
50	ig	ug	2	62	bu	pp	1
50	il	ul	2	62	du	tt	1
50	ip	up	2	62	u	NULL	1
50	ir	ur	2	62	ux	c	1
50	oyy	Ov	1	62	ux	iy	2
50	s	c	1	62	x	Mc	2
50	s	y	1	62	x	c	1
50	y	v	1	62	x	cc	1
50	yy	v	1	62	x	y	1
60	NULL	L	1	70	Av	acc	1
60	NULL	v	1	70	ag	ug	2
60	M	nn	1	70	al	ul	2
60	a	NULL	1	70	ap	up	2
60	ba	pp	1	70	ar	ur	2
60	bu	pp	1	70	av	iy	2
60	da	tt	1	70	av	uc	2
60	du	tt	1	70	d	c	1
60	is	uc	2	70	eyy	Ey	1
60	s	Mc	2	70	iyy	Iy	1
60	s	c	1	70	uv	iy	2
60	s	cc	1	70	v	c	1
60	s	y	1	70	vv	cc	1
60	u	NULL	1	70	y	c	1

60	us	c	1	70	yy	cc	1
60	us	iy	2	71	NULL	c	1
61	M	n	1	71	NULL	v	1
62	NULL	L	1	71	E	ecc	1
62	NULL	v	1	71	O	on	3

Stem Recoding Rules for Rule-based Recoding

Compiling a set of sequential context-dependent transformational rules is not a trivial job. This involves examining several thousands of cases and requires language expertism. A sample (incomplete) set of rules have been compiled for Telugu verbs based on [27, 28]. This small list of rules covered lot many cases. The number of rules required for recoding based on ordered-sets of rules is too less compared to those required for recoding by stem ending replacement. But, at the same time, these rules are difficult to compile.

```
% -----
%      A sample list of stem recoding rules for Telugu Verbs
% -----

%%
~(#{a}##$      : u      : 1, 2      : 070, 071
~(#{u}##$      : u      : 1, 3      : 070, 071
~(#{a}##$      : u      : 1, 1      : 060, 062
**
{u}$           :      : 1, 1      : 060, 061, 062
**
{d}$           : t      : 1, 1      : 060, 061, 062
{b}$           : p      : 1, 1      : 060, 061, 062
**
{S}$           : s      : 1, 1      : 050
**
{s}$           : c, cc, Mc : 1, 1      : 060
**
i{c}$         : Mc      : 1, 1      : 060, 061, 062
{c}$          : c, cc    : 1, 1      : 060, 061, 062
{p}$          : pp      : 1, 1      : 060, 061, 062
{t}$          : tt      : 1, 1      : 060, 061, 062
{L}$          : LL      : 1, 1      : 060, 061, 062
{n}$          : nn      : 1, 1      : 060, 061, 062
**
([a|i|u|e|o]){N}$ : n      : 1, 1      : 060, 061, 062
**
~(#{i}##$      : u      : 1, 1      : 050
**
{x}$          : c, s      : 1, 1      : 062
```

APPENDIX C

Important Modules

This appendix provides some important modules part of the implementation. The modules are discribed in a pseudo-language.

C.1 Suffix Trie Routines

We start with looking the trie data structure used for the suffixes and the insertion and searching routines. Trie is represented in the program by a two dimensional array short integers(2 bytes each). Each row corresponds to one node in the trie and each column represents an input character. An additional column (0th) is used which contains useful information only for the 'external' nodes. For each external node, the path from root to it denotes a valid suffix. The entry in the 0th column of that node gives the category-code of the suffix.

The suffix trie insertion and searching algorithms are as follows. The array `ALPHA_MAP` maps the characters `a` to `z` and `A` to `Z` onto the integers 1 to 52 sequentially.

```
cons ALPHABET_SIZE = 52
cons NULL_LINK = MAX_UNSIGNED_SHORT_INTEGER
type trie_t = array[ALPHABET_SIZE + 1] of unsigned short integer

% Inserts a given suffix into the given reverse suffix trie. The trie array
% elements are all initially set to NULL_LINK. The given trie array should
% contain sufficient space for additional nodes required to insert the suffix.

proc AddTrie(alt trie_io; fix suff_i, catCode_i)
  trie_io : array[] of trie_t      % trie into which to insert
  suff_i : string                  % the suffix to be inserted
  catCode_i : unsigned short integer % category code of the suffix
begin
  i, c : integer
  cur, next : unsigned short integer
  revs : string

  cur = 0
  revs = reverse(suff_i)
  for i :∈ 0 to length(revs) - 1 do
```

```

    c = ALPHA_MAP[revs[i]]
    next = trie_io[cur][c]
    if next = NULL_LINK then
        free = next free location in trie_io
        trie_io[cur][c] = free
        next = free
    fi
    cur = next
rof
if trie_io[cur][0] ≠ NULL_LINK then
    print "duplicate suffix"
else
    trie_io[cur][0] = catCode_i
fi
return
end

```

% Searches a given reverse suffix trie for a given suffix. If found, sets catCode_o
 % to the category-code of the suffix; otherwise, to NULL_LINK

```

proc SearchTrie(fix trie_i, suff_i; alt catCode_o)
    trie_i : array[] of trie_t      % trie in which to search
    suff_i : string                 % the suffix to be searched for
    catCode_o : unsigned short integer % category code of the suffix if found
begin
    i, c : integer
    cur : unsigned short integer
    revs : string

    cur = 0
    revs = reverse(suff_i)
    for i :∈ 0 to length(revs) - 1 do
        c = ALPHA_MAP[revs[i]]
        cur = trie_i[cur][c]
        if cur = NULL_LINK then
            catCode_o = NULL_LINK
            return
        fi
    rof
    if trie_i[cur][0] ≠ NULL_LINK then
        curCode_o = trie_i[cur][0]
    else
        catCode_o = NULL_LINK
    fi
    return
end

```

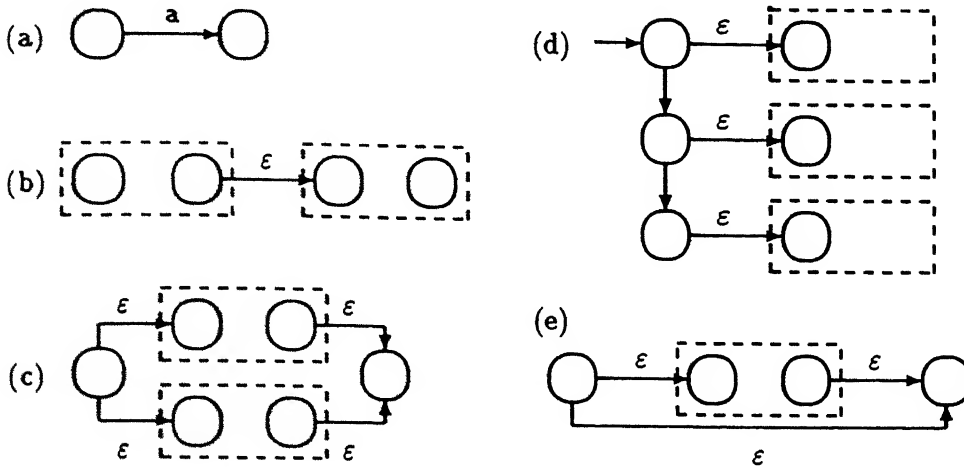



Figure 16: Constructing an NFA from Input Rules Specification

C.2 NFA Construction

A scheme similar to Thompson's construction[1, 3] is used for building nondeterministic finite automata from the regular expression like specification of the input recoding rules. It works as follows:

A single character is the simplest possible expression which can be represented by a correspondingly simple NFA as shown in Figure 16(a).

The concatenation of two expressions e_1 and e_2 is also straightforward – simply connecting the end state of the machine for e_1 to the start state of the machine for e_2 by an ϵ -transition, as shown in Figure 16(b). This method needlessly wastes states, however. A better solution merges the ending state of the first machine with the start state of the second one.

There are two situations in the input rules where an OR of two expressions is required. The first is the OR operator (the vertical bar) itself, which appears in an expression. The OR operator is processed by constructing the machine shown in Figure 16(c). The seemingly empty boxes in the picture represent machines for the two expressions e_1 and e_2 which are combined as shown to form a machine representing $e_1|e_2$.

The second OR situation is the input specification itself. In the input, each rule-group contains many rules, which are effectively ORed together, because any one of the given patterns is searched for in the input word. This high-level OR is done using the scheme shown in Figure 16(d). Each of the boxes is an NFA that represents an entire pattern, and all of these are connected together using several dummy states and ϵ -transitions.

Similarly, machine to recognize the optional operator () is shown in Figure 16(e).

The metacharacters \sim \$ # and @ are also directly used to label the edges for the sake of simplicity of implementation of both the table generator and the analyzer parts.

C.3 NFA to DFA Conversion

The method used for constructing a set of tables representing DFAs from the NFAs is called *subset construction*[1, 3]. Let's first look into some definitions:

ϵ -closure : Given a set of states S , the ϵ -closure set is the set of all states that can be reached by making ϵ -transitions from any state in S . This set also includes the states from S .

Move : Given a set of states S and an input character c , $\text{Move}(S, c)$ is the set of all states that can be reached by making transitions on c from any state in S .

Following are the algorithms that compute ϵ -closure and Move sets. The data structure used for NFA states is as given on page 35.

NFA_g : array[**MAX_NFA_STATES**] of **nfa_t** % the global NFA states array

% Computes ϵ -closure set of a given set of NFA states; and outputs the
 % fields **replace**, **strtPos**, **endPos** and **catCodes** associated
 % with the accepting state in the resulting set.

```

proc EClosure (fix S; alt E, strtPos, endPos, replace, catCodes)
  S : set of integer           % a set of NFA states
  E : set of integer           % the  $\epsilon$ -closure set of S
  strtPos, endPos : integer    % the values of fields associated with
  replace : string             % the accepting state (if any) in the
  catCodes : list of integer    % resulting set E
begin
  st : stack of integer        % a stack for computation
  i : integer                  % the state under consideration
  j : integer                  % last accepting state seen
  n : integer                  % state having an  $\epsilon$  transition from i

  % push all the elements of S onto st
  st = empty
  for each i in S do
    push(st, i)
  rof

  E = S
  j = -1
  while st  $\neq$  empty do
    i = pop(st)
    if NFA_g[i].edge = EMPTY then
      % i is an accepting state; remember it
      j = i
    fi
    if NFA_g[i].edge = EPSILON then
      n = number of the state NFA_g[i].next
      if not member(E, n) then
        addset(E, n)
        push(st, n)
      fi
    fi
    if NFA_g[i].next2  $\neq$  NULL then
      n = number of the state NFA_g[i].next
      if not member(E, n) then
        addset(E, n)
        push(st, n)
      fi
    fi
  end while
end proc

```

```

        fi
    fi
    elihw

    if j  $\neq$  -1 then
        strtPos = NFA_g[j].strtPos
        endPos = NFA_g[j].endPos
        replace = NFA_g[j].replace
        catCodes = NFA_g[j].catCodes
    fi
    return
end

```

% Computes the Move set for a given set of NFA states on a given input character

```

proc Move (fix S, c; alt M)
    S : set of integer          % a set of NFA states
    c : char                    % an input character
    M : set of integer          % the Move set
begin
    i : integer                 % the state under consideration
    n : integer                 % state having an  $\epsilon$  transition from i

    M = empty
    for each i  $\in$  S do
        if NFA_g[i].edge = c then
            n = the state number of NFA_g[i].next
            addset(M, n)
        fi
    rof
    return
end

```

The algorithm for converting NFAs to DFAs is given below. This algorithm calls the EClosure and Move algorithms.

```
const FAILURE_TRANSITION = -1
```

```

% One row of the DTrans matrix; MAXX_CHARS elements for transitions
% on characters and 1 for marking wether this is an accepting state, if so, pointer
% to the corresponding entry in AStates
type row_t = array[MAXX_CHARS + 1] of integer

```

```
% Structure definition for DFA states during the computation.
```

```

type dfa_t = record
    mark : boolean              % for marking the states; see below
    set : set of integer        % set of NFA states represented by this state
    replace : string            % these four fields used for accepting
end record

```

```

    strtPos : integer          % states only; they are copied from
    endPos : integer           % the corresponding NFA states
    catCodes : list of integer % directly
drocer

```

% Given an NFA state array, build the three tables DStates, DTrans, RStart
 % representing the equivalent DFAs.

```

proc NFAtDFA (fix NFA; alt DStates, DTrans, RStart)
    NFA : array[] of nfa_t      % the input NFA
    DStates : array[] of dfa_t  % the DFA state array
    DTrans : array[] of row_t   % the DFA transition matrix
    RStart : array[] of integer % the rule group starting states
begin
    i : integer                  % index in NFA of the current rule-group start state
    j : integer                  % DFA state currently being expanded
    nstates : integer            % number for the next new state that will be
                                % generated
    S : set of integer           % a set of NFA states that define a DFA state
    c : char                     % a possible input symbol
    rindex : integer             % index into RStart where the next entry will be
                                % inserted

    i = 0
    nstates = 0
    rindex = 0
    DTrans = FAILURE_TRANSITION
    while i ≠ -1 do
        S = {i}
        if NFA[i].rgedge = NULL then
            i = -1
        else
            i = number of the state NFA[i].rgedge
        fi
        call EClosure(S, DStates[nstates].set, DStates[nstates].strtPos,
                     DStates[nstates].endPos, DStates[nstates].replace,
                     DStates[nstates].catCodes)
        DStates[nstates].mark = FALSE
        RStart[rindex++].mark = nstates
        while there is an unmarked DFA state j do
            Dstates[j].mark = TRUE
            for each input symbol c do
                call Move(Dstates[j].set, S)
                if S ≠ empty then
                    if a state with set ≡ ε-closure(S) isn't in DStates then
                        call EClosure(S, DStates[nstates].set,
                                      DStates[nstates].strtPos, DStates[nstates].endPos,
                                      DStates[nstates].replace, DStates[nstates].catCodes)
                        DStates[nstates].mark = FALSE
                        DTrans[j][c] = nstates++
                    else
                        DTrans[j][c] = index of existing state in DStates with

```

```

                                set  $\equiv \epsilon$ -closure(S)
                                fi
                            fi
                        rof
                    elihw
                elihw
            end

```

The `set` and `mark` fields of entries in the `DStates` array are useful only for the purpose of this algorithm. They are no more required after the DFA is built. All other fields of the `DStates` array contain useful information only for accepting states. An additional step of table compression makes use of these observations before the tables are printed on a file. A new data structure `accept_t` is now used, as given on page 36. This is similar to `dfa_t` except that the two fields `mark` and `set` are not there in this. The table `AStates`, an array of `accept_t`, is constructed from `Dstates`, which contains entries only for accepting states.

One extra column is added to the `Dtran` matrix which contains -1 for nonaccepting states and an index into the corresponding entry in `AStates` for the accepting states.

C.4 Running the DFAs

The sequence of machines represented by the three arrays `DTrans`, `AStates`, and `RStart`, are run on the given stem to perform recoding. The algorithm used for this is given below.

% Given a stem after an applicable suffix, whose category-code is also given, is stripped
 % off a word, this algorithm applies the sequence of recoding rules to get all possible
 % roots of the stem.

```

proc RecodeStem (fix stem, cc; alt s1)
  stem : string           % the stem to be recoded
  cc : unsigned short integer % category-code of the removed suffix
  s1 : list of string     % a list of all possible roots
begin
  rgss : integer          % start state of the current rule-group being processed
  ns1 : list of string    % holds the list of altered stems after applying a rule-group
  s : string              % the stem under consideration
  cur : integer           % current state of the DFA
  c : character           % current character in s
  p, q : integer          % pointers into s

  s1 = { stem }
  for (each rgss in RStart in sequence) do
    ns1 = empty
    for (each s in s1) do
      p = 0
      while (s[p]  $\neq$  EOS) do
        cur = rgss
        q = p
        while (s[q]  $\neq$  EOS) do
          if (there is a transition from cur on s[q]) then

```

```

        cur = new state
        if (cur is an accepting state and cc is in
            catCodes associated with cur) then
            perform the recoding associated with cur on s to
                get a list of variated stems ns1
            goto 200
        fi
        q = q + 1
    else
        goto 100
    fi
    elihw
    100 : p = p + 1
    elihw
    200 if ns1 = empty then
        addtolist(ns1,s)
    fi
    rof
    s1 = ns1
end

```

C.5 Recoding by Stem Ending Replacement

In this section, we present the procedure used for recoding by replacing the stem ending. The procedure is straight-forward. The data structure `rcrule_t` used for representing the recoding rules is given on page 37.

`rcrules_g` : array[`MAX_RULES`] of `rcrule_t` % the global rules array

% Given a stem after an applicable suffix, whose category-code is also given, is stripped
 % off a word, this algorithm applies the sequence of recoding rules to get all possible
 % roots of the stem.

```

proc RecodeStem (fix stem, cc; alt s1)
    stem : string           % the stem to be recoded
    cc : unsigned short integer % category-code of the removed suffix
    s1 : list of string      % a list of all possible roots
begin
    root : string           % one possible root
    rn : integer            % current rule number

    s1 = empty
    rn = number of the first rule associated with cc
    while rcrules_g[rn].catCode = cc do
        root = copy(stem_i)
        if rcrules_g[rn].oldEnd ≠ NULL then
            if root ends in rcrules_g[rn].oldEnd then
                strip it off
            end if
        end if
    end while
end proc

```

```

        else
            rn = rn + 1
            continue with next rule
        fi
    fi

    if rcrules_g[rn].newEnd  $\neq$  NULL then
        concat(root, rcrules_g[rn].newEnd)

    if rcrules_g[rn].minRootSize  $\leq$  length(root) then
        addtolist(sl, root)
    fi
end

```

C.6 The Hash Function

The hash function used in the implemented check hash based lexicon is given below. RandHash is a 256-element array of unsigned long's. This array is constructed from four independent permutations of the integers 0 to 255, one permutation is each of the four bytes of the unsigned long.

```

/* Returns the hashcode for a given word word_i.
 * mask_i is the upper bound for the hashcode value.
 */

unsigned long
HashFunction(word_i, mask_i)
register char * word_i;
unsigned long mask_i;
{
    register int c;
    register unsigned long h = 0;
    extern unsigned long RandHash[];

    if ((h = (unsigned long)*word_i++) == 0)
        return(h);

    while ((c = *word_i++) != 0)
        h += RandomHash[ 0xFF & ((h >> 24) ^ c )];

    if ((h = h % mask_i) == 0)
        h = 1;

    return(h);
}

```